REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	4.Jan.00	DISSERTATION

4. TITLE AND SUBTITLE
A DATA COLLECTION AND REPRESENTATION FRAMEWORK FOR SOFTWARE AND HUMAN-COMPUTER INTERACTION MEASUREMENTS

5. FUNDING NUMBERS

6. AUTHOR(S)
MAJ MATHIAS KARL S

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
AUBURN UNIVERSITY MAIN CAMPUS

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
THE DEPARTMENT OF THE AIR FORCE
AFIT/CIA, BLDG 125
2950 P STREET
WPAFB OH 45433

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
FY99-633

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION AVAILABILITY STATEMENT
Unlimited distribution
In Accordance With AFI 35-205/AFIT Sup 1

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

14. SUBJECT TERMS

15. NUMBER OF PAGES
169

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT

DISSERTATION ABSTRACT

A DATA COLLECTION AND REPRESENTATION FRAMEWORK FOR SOFTWARE AND HUMAN-COMPUTER INTERACTION MEASUREMENTS

Karl Steven Mathias

Doctor of Philosophy, December 11, 1999
(M.S., Air Force Institute of Technology, 1993)
(B.S., Utah State University, 1986)

Directed by James H. Cross II

This dissertation presents a framework for capturing, storing, and correlating human-computer interaction measures and software attribute measures. Discovering how computer programmers comprehend software has long been a problem in the field of computer science. Researchers want to understand what strategies programmers take as they look at software so that superior strategies can be investigated and documented. This involves collecting interaction and comprehension measures. Industry wants to determine when software will cause the greatest problems in comprehension, and if the software can be changed to ease this learning curve. This involves the collection of software attribute measures.

A framework for the collection and correlation of both measures was developed based upon a straight-forward mathematical model. In this model, a user action is combined

with the state of the software being manipulated at a specific point in time. This combination of action, software state, and time forms a unique tuple. This tuple is then the core element of analysis procedures which determine how software state affects user actions. From a mathematical standpoint, the framework components maintain sets of these tuples, organized by user session.

The framework was designed and implemented using object-oriented techniques. It consists of a several key subsystems: the development environment, a set of core data collection components, and an analysis tool. The core components were the focus of the research, and consist of Java classes which handle event collection, storage, and extraction for analysis in a portable manner.

The framework was tested by integrating it with a development environment for Microsoft Windows 98 called pcGRASP. To analyze the results of data generated by pcGRASP, an analysis tool was developed called the Collection Framework Analyzer (CFA). The framework was then tested to ensure that data generated by pcGRASP was stored accurately by CFA, that large datasets could be stored, that software attribute information was correctly handled, and that framework data was portable regardless of the host operating system that it was originally generated under.

The Framework gives researchers and industrial users a method for conducting long-term studies of what makes a development environment effective, and how to maximize comprehension in hopes of lowering defect rates in software.

A DATA COLLECTION AND REPRESENTATION FRAMEWORK

FOR SOFTWARE AND HUMAN-COMPUTER

INTERACTION MEASUREMENTS

Karl Steven Mathias

Certificate of Approval:

Kai-Hsiung Chang Professor Computer Science and Software Engineering	James H. Cross II, Chairman Professor Computer Science and Software Engineering
T. Dean Hendrix Assistant Professor Computer Science and Software Engineering	John F. Pritchett, Dean Graduate School

A DATA COLLECTION AND REPRESENTATION FRAMEWORK FOR SOFTWARE AND HUMAN-COMPUTER INTERACTION MEASUREMENTS

Karl Steven Mathias

A Dissertation

Submitted to

the Graduate Faculty of

Auburn University

in Partial Fulfillment of the

Requirements for the

Degree of

Doctor of Philosophy

Auburn, Alabama

December 11, 1999

ACKNOWLEDGEMENTS

I would like to thank my advisor, Dr. James Cross II, for his help in guiding my work on this dissertation. His clear vision on the potential of this research helped my intellectual and professional growth immensely. Sincere appreciation also to my committee members for their time and patience as this work progressed.

My thanks to Mr. Kenneth Lavoie who directed me to this opportunity and pushed me to test my limits. He was, and I suspect always will be, the best boss I've ever had.

My gratitude and love to family and friends who supported me through years of study, work, and twisted schedules.

Finally, to the high school English teacher who made such a key difference in my life, so many years ago: Thank you Mr. Odell, for teaching me that words are so much more than just marks on a page.

Style Manual or Journal used

Association of Computing Machinery Computing Surveys

Computer Software used

Microsoft Word 97

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1. OVERVIEW

1.1 INTRODUCTION

Industry and researchers share a common problem in the realm of empirical studies of programmers. Both require the ability to measure software characteristics, and the ability to measure the effectiveness of the programmer who writes the software. Industry requires it so that it can improve quality and productivity, while researchers need it to understand the software development process. When either attempts to collect measurements using conventional means, they must interrupt the programmer to ask questions. Barring that, they must force the programmer to take unnatural actions such as "thinking aloud" that might influence their behavior. What is needed for both academic and industrial use is the ability to take measurements without intruding on the programmer's thought process. This dissertation presents a solution to this problem in the form of a framework that supports the automatic collection and analysis of human-computer interaction and software measures.

1.2 TRADITIONAL SOFTWARE MEASUREMENT

The field of software measurement has grown up around a desire to improve the delivered software product. Improvements can be considered in terms of the quality and of the cost of the software delivered. Quality is a relative term which will be defined more precisely in the next chapter, but in general it refers to a product which meets user

requirements and is reliable within a given tolerance level. Cost refers to the amount of effort and resources required to produce the product.

Quality and cost factor measurements can be performed directly. For example, if quality were defined as the number of unique failures of a software product during the first year of delivery, then each new failure would be recorded in a log, resulting in a numeric value representing the "quality" of the product. In this case, the quality would be the same as the failure rate. Cost can be measured directly by recording the number of hours spent by staff on a product, multiplied by their salaries, then adding in the dollar cost of resources used such as computing hardware and off-the-shelf software packages used.

What the software engineering community strives to do, however, is to predict the quality and cost of software before and during the development of the software. Prediction implies that properties of the software being developed are analyzed and then key attributes of these properties are used as predictors of the end result. Over the past two decades, several hundred proposed methods for predicting on the basis of attribute measures have been proposed [Zuse 1997]. Many of these proposed methods have never been empirically validated, or have suspect methodology, yet have been used extensively in both industry and in academia [Basili and Weiss 1984, Fenton 1994, Fenton, Pfleeger, and Glass 1994].

1.3 PROGRAM COMPREHENSION

Another area of considerable interest to researchers is program comprehension during development and maintenance of software. The key interest is in learning how programmers understand existing code so that it can be changed, and how programmers use development environments while making the changes. As with software measurement, the human-computer interaction research community sees the improvement of the actual development process itself as a way to improve the quality and lower the cost of the software product.

One of the more interesting areas of study in the program comprehension field is program visualization. Program visualization seeks to show that replacing or enhancing program text with graphical images improves the ability of a programmer to understand the software being worked on. As with software measures, graphic representations are typically static snapshots of the software that are presented to the programmer. Considerable work is being done, however, to study the effectiveness of animated (dynamic) graphic representations of computer code.

Other visualization techniques seek not to replace the program code, but to enhance it. This enhancement may be as simple as formatting the code with indentation, changing the font of the text, or changing the color of the text. More involved techniques overlay the code itself with graphic symbols designed to enhance the understanding of the programmer.

Measuring the effectiveness of these techniques has generally involved gathering empirical measures via experiments. There are several techniques for performing these experiments, but the usual method is to expose a test group to a graphic representation of program code and a control group to the textual representation. Subjects are then tested for comprehension. Another method is to have the groups make changes to code and then compare defect rates.

1.4 THE MEASUREMENT PROBLEM

Software measurement and program comprehension are closely related in several ways. They both seek, indirectly, to improve the quality and cost of software. They make use of measurement as a tool for determining whether progress toward the goal of better software. Empirical studies are performed in both areas to gather these measurements. Both areas examine the attributes of the underlying software as part of the study process.

While there are many tools available for collection of static software measurements, there are very few available to assist in the study of program comprehension. Tools that do exist are usually special-purpose devices applicable only to the field of program comprehension and not generally suitable for industrial use. There were no tools found which collect and correlate data for both static software measurements and program comprehension.

The lack of tools to collect this data presents a problem for a company or research group which creates a tool for developing software. For example, they may create an integrated development environment (IDE) for producing Java code. Or they may

develop an environment for creating structure diagrams that can assist the programmer with automatically generated code. However, without a way to measure how the programmers use these devices, and the effect of that use on the software being developed, it is difficult to determine how effective the environment is.

Another problem with measurement involves human factors. Programmers generally want to write code and minimize any administrative details that go along with that task. Managers in companies, researchers on projects, and programmers following the personal software process (PSP) [Humphrey 1997] require measurements to determine progress, quality, and cost. However, if measurement requires the programmer to stop and make a manual entry, either on paper or on a computer form, then their task is interrupted. The effect of this is that programmers begin to avoid the task, or do not put down accurate information [Perry *et al.* 1994, Basili and Weiss 1984]. Because of this, mechanisms are needed that minimize the impact on them.

1.5 A DATA COLLECTION FRAMEWORK

This dissertation presents a solution to these problems in the form of a data collection framework. The framework is a flexible set of software components that can be used by development environments for collecting, storing, organizing, and analyzing programmer and software measurements. The framework's architecture makes it easily adaptable to different environments, and the implementation of the framework in Java makes it portable across many platforms.

It is a relatively simple task to write a routine in an environment to collect and display summary information. What makes this framework unique is its ability to collect information and show not just a summary, but a timeline of user actions and software changes as they occurred. For example, instead of just knowing how many compiles a user made, it is possible to determine when the compiles occurred and the state of the software before each compile. As will be shown in Chapter 3, this is an important requirement for meaningful analysis.

The collection framework assumes three major systems exist. The first system is the development environment or tool. The second are the collection framework core components, which are used by the environment to store information. The third system is the analysis back-end tool which extracts the information from the framework and performs analysis and display. As part of this work, a package of Java utilities were developed to assist in the creation of meaningful reports and charts for the analysis system.

1.6 RESEARCH WORK

This document reports on the investigation which lead to the framework being developed and implemented. A primary concern during the investigation was ensuring the design of the framework would support the goals of GRASP (Graphical Representations of Algorithms, Structures and Processes). The GRASP research project at Auburn University required a tool to collect software and usage metrics for the combined study of program complexity and software visualization. In order to determine how programmers

use the Control Structure Diagram (CSD) in their programming process, GRASP had to be able to record their actions. Since there could be several hundred subjects participating, the data must be collected and stored in a central repository for later analysis.

Using object-oriented techniques combined with a relational database storage mechanism, the framework was successfully constructed and integrated with pcGRASP. Test sessions were conducted to show that data could be successfully collected in real time from pcGRASP without impacting the programmer. This same data was successfully extracted and analyzed using a graphical analysis tool built for the GRASP project as part of this dissertation. Test sessions were also conducted to ensure that the framework and analysis tool could operate on multiple operating systems, without regard to the host operating system of the development environment supplying the data. Specifically, the system was successfully tested under Linux and Windows 98.

1.7 ORGANIZATION OF THE DISSERTATION

The dissertation is organized into six chapters. Chapter Two presents a background investigation into current research of software metrics, program comprehension, and software visualization. Chapter Three presents a detailed analysis of the problem with collecting and analyzing measurements of this data. Chapter Four is a detailed description of the design rationale behind the framework. Chapter Five describes how the framework was tested by enhancing pcGRASP and developing a graphical analysis

tool. Chapter Six discusses the results of the framework's integration, the benefits of this approach, and future work to be conducted in this area.

CHAPTER 2. LITERATURE REVIEW

2.1 INTRODUCTION

This section will look at some of the work that has been done in areas that bear on the problem of tying software measurements to program comprehension. Since the goal is to improve software quality and lower cost, discussion will start by examining what "quality" means in relation to software. Next, the field of software measurements will be examined. This is followed by a discussion on the work being done in program comprehension. The review concludes with a look at the current state of work being done in automatic data collection for usability studies.

2.2 SOFTWARE QUALITY

The purpose of software measurement is to assist in tracking and improving software quality. In order to understand whether this is successful, it is essential the term *software quality* be carefully defined. As starting point for looking at the generic term *quality*, one can look at a standard definition such as that contained in *The American Heritage Dictionary*:

> Quality. 1. The essential character of something; nature. 2. A characteristic or attribute; property. 3. Degree of excellence. 4. High social position. [Heritage 89]

This can be applied to software by measuring the characteristics and attributes of software development artifacts. The question that software engineers struggle to answer is: "What attributes of software should be measured to determine the existence of high or low quality?" Even before that discussion can begin, it is necessary to define what is "high" and what is "low" in terms of quality. This is usually done in terms of certain key factors. Three major definitions of what factors determine software quality will be examined.

2.2.1 McCall's Quality Factors

In 1976, a study was conducted on behalf of the United States Air Force's Rome Laboratory. The purpose of this study was to create a handbook for Air Force acquisition managers that would guide them in specifying the overall quality of a software system in contracts [Fedchak and Vienneau 1996]. The result of this study was a set of guidelines known within the Air Force as the Rome Laboratory Software Quality Framework (RLSQF). RLSQF was created by James McCall and Gene Walters and contained one of the first, and most widely quoted, formalizations of the term software quality.

McCall [McCall *et al.* 1977] proposed a set of key software quality factors that affect software quality (as listed in [Pressman 1997]):

- *Correctness.* The extent to which a program satisfies its specification and fulfills the customer's mission objectives.
- *Reliability.* The extent to which a program can be expected to perform its intended function with required precision.

- *Efficiency.* The amount of computing resources and code required by a program to perform its function.
- *Integrity.* The extent to which access to software or data by unauthorized persons can be controlled.
- *Usability.* The effort required to learn, operate, prepare input, and interpret output of a program.
- *Maintainability.* The effort required to locate and fix an error in a program.
- *Flexibility.* The effort required to modify an operational program.
- *Testability.* The effort required to test a program to ensure that it peforms its intended function.
- *Portability.* The effort required to transfer the program from one hardware and/or software system environment to another.
- *Reusability.* The extent to which a program [or parts of a program] can be reused in other applications.
- *Interoperability.* The effort required to couple one system to another.

2.2.2 Boehm's Quality Model

Boehm conducted a study of quality factors at TRW. In the Boehm quality model, the following are proposed as key quality factors (as defined in [Boehm *et al.* 1978]):

- *Understandability.* A software product possesses this characteristic to the extent that the purpose of the product is clear to the evaluator.

- *Portability.* Can the product stand by itself, or to what extent does it require additional resources.
- *Maintainability.* A product has this characteristic to the extent that it facilitates updating to satisfy new requirements.
- *Testability.* A software product possesses the characteristic testability to the extent that it facilitates the establishment of acceptance criteria and supports evaluation of its performance.
- *Usability.* A software product possess this characteristic to the extent that it is convenient and practicable to use.
- *Reliability.* A product has this characteristic to the extent that it can be expected to perform its intended functions satisfactorily.
- *Efficiency.* A software product possesses efficiency to the extent it fulfills its purpose without waste of resources.

2.2.3 FURPS

Another set of factors is that used by Hewlett-Packard. FURPS (Functionality, usability, reliability, performance, supportability) is part of a company-wide effort to establish a metrics program [Grady and Caswell 1987].

The FURPS factors are:

- *Functionality.* The product's feature set, capabilities, generality, and security determine this factor.

- *Usability.* This factor is determined by the product's human factors, aesthetics, consistency, and documentation.
- *Reliability.* The product's frequency/severity of failure, recoverability, predictability, accuracy, and mean time to failure.
- *Performance.* The product's speed, efficiency, resource consumption, throughput, and response time.
- *Supportability.* Determined by testability, extensibility, adaptability, maintainability, compatibility, configurability, serviceability, installability.

After years of use at Hewlett-Packard, FURPS retains its usefulness. Grady attributes some of this success to the fact that the acronym itself is easy to remember. The system currently in use is FURPS+ which reflects some minor additions to the definition, most notably that of "localization" which refers to the ease with which a software system may be moved to different world locales where English may not be the predominant language or system of measurement [Grady 1992].

2.3 SOFTWARE MEASUREMENT CONCEPTS

In order to determine whether the quality of software is improving, it is necessary to measure. The question is: what should be measured?

To answer this, the general model for software measurement uses the following terms:

Entities. In terms of software, these are the products, processes, or resources that are observed and manipulated in some manner [Kitchenham *et al.* 1995]. For

example, a program executable is an example entity that can be observed, and may be manipulated in some manner such as reducing the number of times it fails.

Attributes. Attributes are properties of an entity that can be measured. By "measure" the implication is that some relationship can be captured between an attribute in the real world and its corresponding value in the mathematical world [Kitchenham *et al.* 1995]. A mapping of the attribute "run time" to the set of real numbers would be an example of this type of relationship.

For software, attributes can be further broken down into those that are internal and those that are external [Fenton and Pfleeger 1997]. External attributes are measured not by examining the entity itself, but instead by measuring the behavior of the entity. For example, the number of failures in a program executable might be measured. The quality factors discussed in Section 2.1 can be considered external attributes of software products, processes, and resources. The problem with these attributes is that they can typically only be measured at the end of the software development process.

Internal attributes are measured by examining the entity directly, without regard to its behavior. For example, one might measure the size of an executable program in the number of bytes of hard disk space used (assuming a need for such a measurement). An advantage of internal attributes is that many can be measured during development.

The goal is to use internal attributes as predictors of the final quality attributes by defining axiomatic or empirical relationships between them. There are a large number of attributes that can be measured, many thousand potential measures have been proposed. Some potential categories are listed in Table 2-1.

While all of these potential areas of collection hold interest, for the purposes of this research, this discussion will focus on those dealing with the code. The reason for this is that it is necessary to find a method by which measurements can be automatically collected and stored. Specifically, measurements need to be taken as the programmer

Entities	Internal Attributes
Products	
Specifications	Size, reuse, modularity, redundancy, functionality, syntactic correctness, …
Designs	Size, reuse, modularity, coupling, cohesiveness, functionality, …
Code	Size, reuse, modularity, coupling, functionality, algorithmic complexity, control-flow structuredness, …
Test data	Size, coverage level, …
…	
Processes	
Construction specification	Time, effort, number of requirements changes, …
Detailed Design	Time, effort, number of specification faults found, …
Testing	Time, effort, number of coding faults found, …
…	
Resources	
Personnel	Age, price, …
Teams	Size, communication level, structuredness, …
Software	Size, price, …
Hardware	Size, price, speed, memory, …
Offices	Size, temperature, light, …
…	

Table 2-1. Categories of Measures [Fenton97].

develops the code product, with an additional focus on aspects of their usage of the development environment (see Section 3.2).

2.3.1 Code Metrics

This section will consider one narrow aspect of software measurement, that of code measurements. As indicated previously by [Fenton and Pfleeger 1997], there are many measures that can be taken of code attributes. A quick review of the literature generally finds this type of data split into a few categories: size, structure, and measurements unique to the object-oriented paradigm.

2.3.2 Size

Measures of program size are important to industry since they can be used to predict the cost of development. The following sections discuss three methods for measuring size: lines of code, token counting and function point computation.

2.3.2.1 Lines of Code

One of the simplest measurements of size is to count the number of lines of code (LOC) in a unit of software. Watts Humphrey's Personal Software Process (PSP) uses LOC as a predictor of cost in terms of programming effort [Humphrey 1997]. LOC is almost universally discounted as a valid measure quality factors primarily because the number of lines of code can be easily influenced by a specific programmer's style, whether intentional or unintentional [Shepperd and Ince 1993].

Even though LOC is a simple metric to collect, there is great difficulty in defining exactly what constitutes a "line of code." As pointed out by Capers Jones, despite the fact that programmers have been generating lines of code for many decades, there is no true international standard for how to count them [Jones 1991]. The easiest way to count LOC is by simply taking a count of all the physical lines in the source code. This causes problems, however, when considering languages such as Ada and C that allow statements to be split across line boundaries. Style differences will result in different LOC values for programs implementing identical functionality.

Software Productivity Research, Inc. (SPR) has proposed some rules for counting source code [Jones 1991]:

- *Multiple Languages.* Count each language's source separately.
- *Source Code Termination.* Count by statement delimiters, not by physical lines.
- *Source Code Counting.* Count executable statements and data definitions, do not count comments.
- *Macro Instructions and Expansions.* Count all macro instructions and each unique expansion of a macro.
- *Code Categories.* The SPR recommends keeping separate counts for reused code, program generator input/output code, job control language, baseline code, changed code, deleted code, support code, and test code.

In general, it is agreed that LOC is not a good predictor of quality factors and of complexity [Bieman 1996]. It suffers from having a poor definition, and even when a

definition is given, style differences between programmers can produce radically different LOC values, sometimes by a factor of 5 to 1 [Jones 1991].

LOC is useful in one respect, however. It can be used as a baseline against which other measures can be compared for effectiveness [Shepperd and Ince 1993, Basili and Hutchens 1983]. For empirical studies, this gives researchers a "null hypothesis" to work against.

2.3.2.2 Tokens

As an alternative to using LOC as a sizing measure, Maurice Halstead proposed a set of "laws" styled after the laws of thermodynamics in physics [Halstead 1977]. Software science laws are based on the number primitive measures of a program, combined with the ability of the brain to process information. These measures are:

μ_1 = the number of total operators in a program

μ_2 = the number of total operands in a program

N_1 = the total number of operators in a program

N_2 = the total number of operands in a program

Using these primitives, the *length* of a program is defined as:

$N = N_1 + N_2$

The *vocabulary* of a program is defined as:

$\mu = \mu_1 + \mu_2$

The *volume* of a program, which is defined as the number of bits required to represent the program, is:

$V = N \times \log_2 \mu$

Using the length measure, LOC can be estimated by dividing it by a language dependent constant. For example, in machine language where an instruction consists of an operator and an operand the constant is defined as 2. For FORTRAN, it is defined as 7.

Software science is a very controversial set of measures, and there have been many concerns over its validity. Problems exist in the ability among researchers to agree on how to count operators and operands. There are problems in taking the original counting mechanisms to languages other than FORTRAN. There are serious concerns regarding the derivation of the formulas, especially in the application of human cognition concepts. A more complete discussion of these concerns can be found in [Henderson-Sellers 1996] and [Fenton and Pfleeger 1997].

2.3.2.3 Function Points

Another way to look at the problem of determining a program's size is to see things from the point of view of the user. The user is not interested in the amount of code delivered, since the difference between a 10 KLOC (thousand lines of code) program and a 100 KLOC program is immaterial if they accomplish the same purpose. The user is interested in delivered functionality.

In the mid-1970's, A. J. Albrecht at the IBM Corporation realized the mismatch between software units of production, lines of code, and software units of consumption, functions [Jones 1991]. His solution was to invent an abstract concept of software production based on factors of interest to the end user. The function point concept was first published in 1979 [Albrecht 1979], significantly revised in 1984 [Albrecht 1984], and continues to be revised by the International Function Point Users Group (IFPUG).

FUNCTION POINT COMPUTATION PROCEDURE

To compute function points (FP) according to the 1984 method, two steps are required (as shown in [Kan 1995]):

First, the function counts (FC) are computed according to the formula:

$$FC = \sum_{i=1}^{5} \sum_{j=1}^{3} w_{ij} \times x_{ij}$$

Where w_{ij} is a weight factor selected from the following Table 2-2, and x_{ij} is the number of components of the type referenced by w_{ij}.

Next, the FC is adjusted by assessing the impact of 14 system characteristics. Each of the following characteristics are given a rating from 0 (no impact) to 5 (high impact):

Factor	Low Complexity	Medium Complexity	High Complexity
External Input	3	4	6
External Output	4	5	7
Logical Internal File	7	10	15
External Interface	5	7	10
External Inquiry	3	4	6

Table 2-2. Weights for Function Point Components.

1. Data Communications
2. Distributed Functions
3. Performance Objectives
4. Heavily Used Configuration
5. Transaction Rate
6. On-line Data Entry
7. End-User Efficiency
8. On-line Update
9. Complex Processing
10. Reusability
11. Installation Ease
12. Operational Ease
13. Multiple Sites
14. Facilitate Change

The total value adjustment factor (VAF) for FC is computed according to the formula:

$$VAF = 0.65 + 0.01\sum_{i=1}^{14} c_i$$

Where c_i is the impact rating for one of the characteristics above. This yields a VAF in the range 0.65 – 1.35. To compute FP, multiply VAF by FC:

$$FP = FC \times VAF$$

THE SOFTWARE PRODUCTIVITY RESEARCH (SPR) INC. METHOD

In 1985, SPR developed an alternative method for measuring function points. The original factors used in the Albrecht method remain, but the complexity adjustment is greatly simplified. This simplified method has been shown to be accurate, on average, to within 1.5 percent of the Albrecht method [Jones 1991]. The following procedure is used to compute FP, using the SPR method.

First, as in the Albrecht 1979 method, FC is computed according to the following formula:

$$FC = \sum_{i=1}^{5} w_i \times x_i$$

Where w_i is a component weight factor taken from the medium complexity column of Table 2-2, and x_i is the number of those components in the system.

Next, the FC is modified by rating two complexity areas on a scale from 1 (low) – 5 (high). The first area being complexity of the problem (algorithms, calculations), and the second area being complexity of the data (number of variables, structures). VAF is then computed according to the formula:

$$VAF = 0.4 + 0.1(c_1 + c_2)$$

Where c_1 and c_2 are the ratings for problem and data complexities respectively. FP is then computed in the usual manner:

$$FP = FC \times VAF$$

Language	Level	Average Source Statements Per Function Point (ASSFP)
Assembly Language	1.0	320
Ada95	6.5	49
C	2.5	128
C++	6.0	53
Java	6.0	53
VHDL	17.0	19

Table 2-3. SPR Programming Languages

SPR BACKFIRE

SPR uses their method of computing function points as a way to also automatically estimate the size of the resulting source code. This has been done by empirically evaluating several projects at differing levels of complexity to determine the average number of statements required per function point for those projects. Based on this information, SPR has rated different languages assigning each an average number of statements required for that language to implement a function point. Statements are counted using the method described in Section 2.2.2.1. A portion of SPR's 1996 Programming Language's Table [Jones 1996a] is shown in Table 2-3.

Note that in Table 2-3, assembly language is considered the baseline language. To determine a language's level, 320 (the ASSFP for the baseline), is divided by the ASSFP for the other language. SPR uses this when comparing similar languages where empirical data may not yet be available. For example, note the similar values of C++ and Java in Table 2-3.

To estimate a project's final LOC, FP is computed as shown previously. The target language is then selected and the ASSFP is multiplied by the project's FP to determine LOC.

Using the same principle, the LOC for an existing software system can be used to estimate its original FP. SPR refers to this estimation as "backfiring" function points [Jones 1991]. Backfiring is accomplished by first collecting the LOC value for the software using the SPR method described in Section 2.2.2.1. FC is then computed as:

$$FC = \frac{LOC}{ASSFP}$$

Next, the user must rate the existing software from 1 – 5 in three categories: problem complexity, code complexity and data complexity. Using these factors, a VAF is computed as:

$$VAF = 0.70 + 0.05(c_1 + c_2 + c_3 - 3)$$

Where c_1 is problem complexity, c_2 is code complexity and c_3 is data complexity. FP is then computed by dividing the VAF into FC:

$$FP = \frac{FC}{VAF}$$

2.3.3 Structural Measures

Size measures are generally used as indicators of how much effort was required to produce a program. Structural measures examine how code is put together and are used to predict the quality of the software. The following sections discuss three methods of

measuring structure: cyclomatic complexity, design complexity and the complexity profile.

2.3.3.1 Cyclomatic Complexity

In 1976, Thomas McCabe proposed a measure of software complexity [McCabe 1976]. Making extensive use of graph theory, McCabe posited that applications with a large number of possible control paths would be more complex to understand, more difficult to maintain, and more difficult to test. The idea being that if two programs A and B have numbers of control paths such that A's count is larger than B's count, then A is considered more complex than B.

The problem with computing control paths is that if a program has a backward branch in its path, then the number of paths a program can take is potentially infinite. Cyclomatic complexity, therefore, only considers the basis paths in a flow graph. That is, independent linear paths are counted without consideration to how many times or in what combination those paths may be executed. This counting may be performed by three methods.

Method 1: The cyclomatic number of a graph G with n vertices, e edges, and p being the number of unconnected parts of the graph, is computed as:

$$V(G) = e - n + 2p$$

Method 2: The cyclomatic number of a graph with π decision predicates may be computed as:

$$V(G) = \pi + 1$$

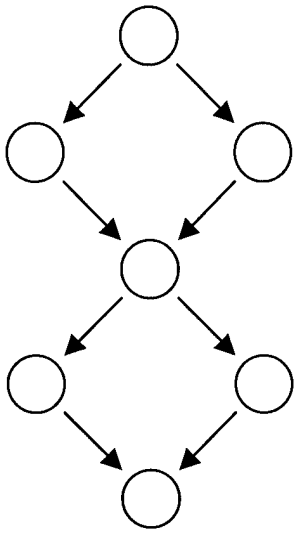


Figure 2-1. Program Control Flow Graph

Method 3: The cyclomatic number of a graph may be computed by counting its regions r:

$V(G) = r$

In Figure 2-1, n is 7, e is 8, p is 1, π is 2, and r is 3. Therefore:

$V(G) = e - n + 2p = 8 - 7 + 2 \cdot 1 = 3$
$V(G) = \pi + 1 = 2 + 1 = 3$
$V(G) = r = 3$

McCabe recommends that in order to have good test coverage and to be maintainable that programmers keep module complexity V(G) less than 10. Numerous studies have been conducted that tend to show a good correlation between defect rates and cyclomatic complexity [McCabe and Butler 1989]. According to Stephen Kan [Kan 1995], studies have also shown a strong correlation between complexity and program size in LOC. When program size is controlled, the correlation coefficient tends to weaken. According

to Kan, however, this may not be accurate since analysis using Pearson's correlation coefficient are sensitive to extreme data points and noise. He recommends use of rank-order correlation coefficients that tend to show a good correlation between complexity and defect rates even when program size is controlled.

2.3.3.2 Design Complexity

In 1989, McCabe and Butler extended the concept of complexity computation into the area of design [McCabe and Butler 1989]. McCabe noted that one of the problems with using cyclomatic complexity based on analysis of existing code was that no predictive measures could be taken to prevent a module or design from becoming overly complex and consequently difficult to maintain and test.

McCabe's design complexity measures are based on analysis of a software system's call structure. The design complexity of a module, defined as $iv(G)$ is determined by reducing the module's flowgraph such that subgraphs which do not influence the module's interaction with other modules are eliminated. These reduced module complexities can then be used to determine the design complexity of a structure chart in the following manner:

$$S_0 = \sum_{i \in D} iv(G_i)$$

Where D is the set of modules which are descendents of a module M, unioned with M. The number of integration tests required is then computed as:

$$S_1 = S_0 - n + 1$$

Where S_0 is the design complexity for the system, and n is the number of modules in the system. An example computing $iv(G)$, S_0, and S_1 is shown in Figure 2-2 taken from [McCabe and Butler 1989]. Note that a module is conditionally executed if there is a black dot on the calling module's edge.

2.3.3.3 Complexity Profile Metric

One of the problems, and major criticisms, of using cyclomatic numbers as a measure of software complexity, is that it produces just a single value to describe a module's complexity. Fenton especially criticizes this approach since it attempts to quantify, in a single value, a "poorly understood, but intuitively recognisable attribute." [Fenton 1994]

An alternative proposed by Patricia McQuaid is a fine-grained approach to computing and visualizing complexity [McQuaid 1996]. McQuaid regards complexity as psychological in nature. The method defines complexity in terms of a program statement's content, much like Halstead's effort measurement, and in terms of context

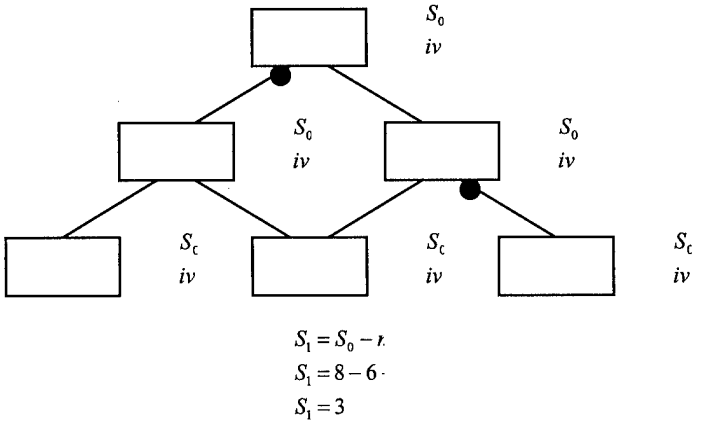


Figure 2-2. Design Complexity Example [McCabe89]

which is the environment in which the statement occurs. Due to the tedious and lengthy nature of the discussion of the Profile Metric, the details of its computation will not be presented here.

What makes the complexity profile metric a particularly interesting measure is that it is designed to offer users the ability to customize it to their requirements. Users select appropriate terms and algorithms, and then each statement's complexity is graphed in histogram form with the vertical axis being complexity and the horizontal axis being the range of statements. The net effect is that areas of high complexity tend to appear as "clusters" of complexity and can be evaluated. This crossing of software measurement with program visualization techniques provides a very powerful capability to users [Cross *et al.* 1997]. An example of a complexity profile is shown in Figure 2-3.

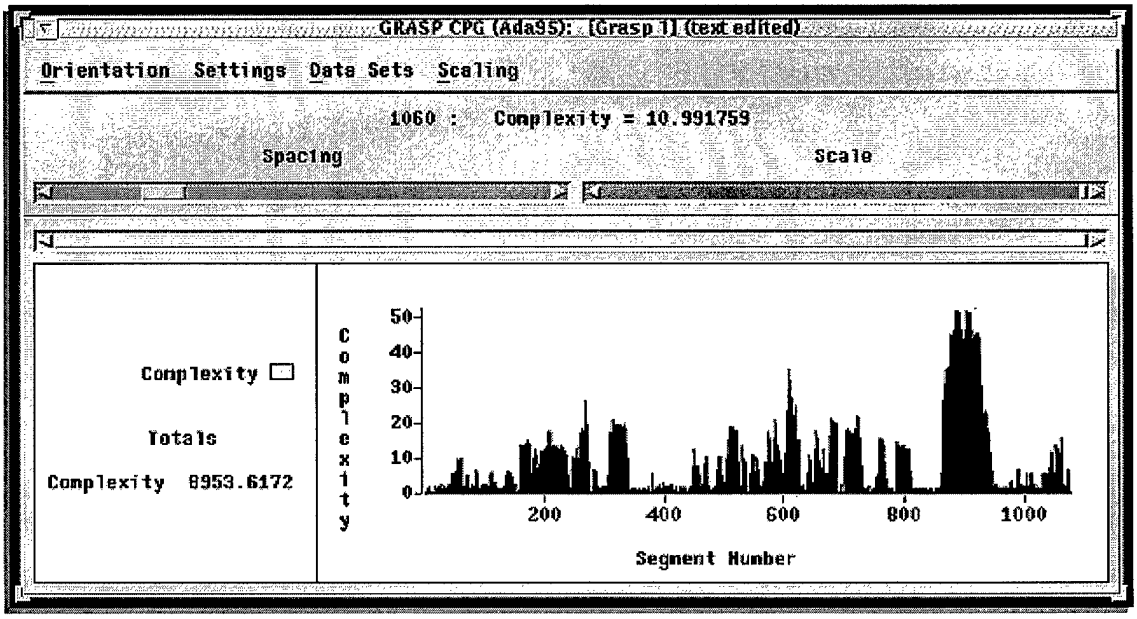


Figure 2-3. Complexity Profile Graph [Cross97]

2.3.3.4 Object-Oriented Measurement

Previous discussion has largely ignored the paradigm under which the target code was developed. However, the object-oriented development methodology introduces new constructs and methods of programming. The following sections discuss proposed measures for object-oriented software.

CHIDAMBER AND KEMERER

Most of the above metrics are designed to deal with traditional, structured-coding techniques. Chidamber and Kemerer developed a set of six design metrics for projects using object-oriented development [Chidamber and Kemerer 1994]. These metrics were developed using, interestingly enough, an ontology from a treatise on philosophy as a theoretical basis. As a more mathematically formal justification for the measures, Chidamber and Kemerer also evaluated their metrics against Weyuker's proposed framework [Weyuker 1988]. The authors also claim to have validated their metrics by developing tools that were used on site at two companies. Their validation, however, is not against defect rates, but merely seems to ensure that the measures were indeed "measured." However, Basili conducted empirical studies of the measures as fault predictors and found good correlation with the exception of the LCOM measure [Basili *et al.* 1996].

Weighted Methods Per Class (WMC)

The first metric presented by Chidamber and Kemerer is based on a belief that classes with large numbers of methods, and/or methods that are complex, require more effort to develop and maintain. Therefore, the WMC metric is computed by summing the complexities of the methods in the class:

$$WMC = \sum_{i=1}^{n} c_i$$

Where c_i is the complexity of method i. It should be noted that the authors do not specify the complexity measure to be used, and indicate that any "traditional" static method may be appropriate.

Depth of Inheritance Tree (DIT)

The next metric is based on the view that the deeper the class is in a hierarchy of classes, the greater number of methods it will inherit. This increases the design complexity of the class. DIT is measured by counting the number of classes in the maximum path taken from the root of the inheritance tree down to (but not including) the subject class.

Number of Children (NOC)

NOC is defined as the number of immediate subclasses that inherit from a class in a class hierarchy. The idea is that if a class has a large number of immediate children, then it will have a larger influence on the system's design and will make testing more complex. That authors also believe that if a class has a large number of children, it may also

indicate improper abstraction. It may also simply indicate a greater amount of reuse in the system.

Coupling Between Object Classes (CBO)

The CBO measures the amount of coupling between object classes. Chidamber and Kemerer define coupling to be the use of another object's methods or instance variables. Since this creates dependencies between objects, it is detrimental to modular design, makes reuse more difficult, and makes testing and maintenance more complex. The CBO count for a class is defined as the count of classes to which it is coupled.

Response for Class (RFC)

The typical method of communication between classes in an object-oriented system is through message passing. The basis for the RFC measure is that if a class responds to a message by calling a large number of methods, it will be more complex and will be more difficult to test and debug. RFC is defined as:

$$RFC = |RS|$$

Where RS is the response set for the class, defined as:

$$RS = \{M\} \cup \{R_i\}$$

Where $\{R_i\}$ is the set of all methods called by method i, and $\{M\}$ is the set of all methods in the class. Note that the set of methods called includes methods outside of the class.

Lack of Cohesion Metric (LCOM)

The final metric proposed by Chidamber and Kemerer measures the lack of cohesion in a class. LCOM is essentially a check of methods against instance variables. If a class's methods use a lot of common instance variables, then the class has good cohesion. If the methods tend not to share instance variables, then the class will have low cohesion and will be more complex.

LCOM is defined by Chidamber and Kemerer in the following manner:

Let a class C have n methods $M_1, M_2, ..., M_n$. Let $\{I_j\}$ be the set of instance variables used by method M_j.

There are n such sets $\{I_1\}, ..., \{I_n\}$. Let $P = \{(I_i, I_j) \mid I_i \cap I_j = \emptyset\}$ and $Q = \{(I_i, I_j) \mid I_i \cap I_j \neq \emptyset\}$. If all n sets $\{I_1\}, ..., \{I_n\}$ are $\emptyset$ then let $P = \emptyset$. LCOM is then:

$$\begin{aligned} LCOM &= |P| - |Q|, \text{ if } |P| > |Q| \\ &= 0 \text{ otherwise} \end{aligned}$$

A class with a higher LCOM value will therefore have more methods that do not share instance variables (the $I_i \cap I_j = \emptyset$ term of P). Higher values of LCOM indicate lower cohesiveness.

LORENZ AND KIDD

Lorenz and Kidd also proposed a set of metrics for object-oriented projects and systems [Lorenz and Kidd 1994]. The set of metrics proposed appear to be a "grocery list" of

measures that can be taken on object-oriented software and there is no formal justification presented for the measures. They also do not validate the measures in terms of their ability to predict software quality. Some of the more interesting measures are discussed in the following sections.

Number of Message Sends (NOM)

Lorenze and Kidd suggest that as an alternative to using LOC to measure the size of a class method, that the number of messages sent by a method should be counted. The authors note that hybrid OO languages such as C++ would have to be carefully counted since it is possible to have a few message sends surrounded by many lines of non-OO code. The examples presented by the authors show that languages such as Smalltalk seem to benefit the most from this type of sizing.

Average Number of Instance Methods Per Class (NIM_{avg})

The NIM_{avg} measure is determined by averaging the count of all the instance methods in over the number of classes in a software system. Inherited methods are not counted for purpose of this measure. The intention of taking this measure is that if the value is large, then it is possible that too much responsibility is being given to the classes.

Specialization Index (SIX)

Lorenze and Kidd suggest several measures relating to how inheritance affects methods in a class. Generally, these involve counting methods inherited, overridden, etc. The

specialization index is a hybrid measure that is supposed to give a sense of the quality of the subclassing. SIX is defined as:

$$SIX = \frac{NMO \times HNL}{NCM}$$

Where NMO is the number of methods overridden by a class, HNL is the nesting level of the class (same as Chidamber and Kemerer's DIT [Chidamber and Kemerer 1994]), and NCM is total number of class methods.

Average Parameters Per Method (PPM_{avg})

The authors suggest that increasing the number of parameters required to use a method in a class increases the complexity of the coupling between classes. PPM_{avg} is defined as the average number of parameters per method in a class. A high PPM_{avg} indicates that use of the class will increase overall system complexity.

2.4 PROGRAM COMPREHENSION

Many of the metrics described previously refer to the complexity of the software being examined. In the case of McCabe, complexity can be defined as the number of test cases required to fully exercise a software module. More test cases means greater testing complexity. In most cases, however, complexity in the above metrics tends to refer to cognitive complexity. Halstead's software science in particular deals with how programmers are able to capture and manipulate information. Specifically, that there are limits to how much information can be manipulated, and as the need to manipulate more information is required, mistakes will be made.

This section addresses research into the ability of programmers to comprehend code. First, a general introduction to some basic research into this area is presented. Next, a more specific discussion is given on how visualizations of software impact comprehension.

2.4.1 Models for Program Comprehension

The three main strategeies that have been identified for comprehending software are top-down, bottom-up, and mixed model. The following sections describe each of these strategies in terms of the research that was done to identify them.

2.4.1.1 Soloway

The idea that complexity is related to cognitive ability is supported by work conducted by Soloway, Bonar, and Ehrlich in studying how programmers comprehend looping constructs [Soloway *et al.* 1983]. They found that programmers develop preferred strategies for certain types of loops. Preferred strategies required fewer cognitive elements to understand, and when programmers deviated from using these simpler strategies they made more errors.

Soloway later extended the idea of looping strategies into a more general programming comprehension model of *goals* and *plans* [Soloway 1986]. In this model, the programmer is given a set of requirements for a problem. The programmer mentally converts these requirements into sets of goals. Goals are broken down into subgoals until a level is achieved at which a plan can be applied to a goal. A plan is a "canned" solution

for a certain type of problem. For example, Soloway offers the following as an example plan for keeping a running total:

```
Initialize a running total
Ask user for a value
If input is not the sentinel value, then
   Add new value into running total
   Loop back to input
```

This plan is instantiated for the language in use. As programmers develop skills, they develop a repertoire of plans that they are able to abut, nest, and merge with each other to solve problems.

Soloway also points out that another way to view goals and plans is to look at plans as the "mechanisms" that are put into the computer to solve the problem. The goals are then viewed as the "explanations" as to why the mechanisms exist.

2.4.1.2 Pennington

Pennington proposes that program knowledge is gained in a more bottom-up manner [Pennington 1987]. In this framework for comprehension, there are two models for representing knowledge: the program model and the situation model. She found that as people examine code, good comprehenders will switch between these models and poor comprehenders will stay with one or the other almost exclusively.

Pennington defines the program model as "a representation that highlights procedural program relations in the language of programs." The situation model is defined as "a

representation that highlights functional relations between program parts that is expressed in the language of the domain world of objects." In other words, the program model explains how specific code constructs work, the situation model explains why the code is there to do that work.

In an empirical study using think aloud techniques, Pennington found that programmers with poor comprehension tended to either get bogged down in the details of the code, or would continuously make hypothesis about the overall working of the system based on code elements. In this respect, they spent most of their time in the program model or in the situation model, with little crossover. Programmers exhibiting good comprehension tended to switch between the models without staying exclusively in one or the other.

2.4.1.3 von Mayrhauser and Vans

von Mayrhauser and Vans studied programmers working with large scale coding projects [von Mayrhauser and Vans 1994, 1996a]. They concluded that program comprehension involves the integration of four model components into a single framework. Their integrated code comprehension model includes a top-down domain model, a program model, a situation model, and a knowledge base. As the authors note, this is basically an integration of Soloway's top-down approach with Pennington's bottom-up approach.

The top-down model is a general understanding of the system and the problem domain of the system. Programmers familiar with a type of problem domain will construct a top-down model of the code. In the example given by von Mayrhauser:

> "For example, a compiler expert can quickly decompose a new compiler into major components such as a lexical scanner, a parser, a code generator, and a code optimizer. Each of these can be further decomposed into component parts e.g., the parser translates expressions into subexpressions, terms, and factors."

According to von Mayrhauser and Vans, the program model is used most when a programmer is new to the code being examined. In this case, domain knowledge is lacking and cannot explain the purpose of the code or what it is doing. The programmer will study the code to determine its control flow, trying to answer questions about what the code is doing. This detailed low-level view of the code is the program model.

The situation model can be seen as a higher-level functional abstraction of the code. It is developed after the program model is in place. Using a bottom-up approach, the developer takes portions of the program model and chunks it into a more abstract concept that fits within the situation model. For example, discovering the implementation of a hash table (program model) is where the compiler symbol table is stored (situation model).

von Mayrhauser and Vans found that development of these models can occur in two ways: systematic or opportunistic. A systematic strategy involves moving methodically through the code, building program and situation models as progress occurs. Top-down model development is more opportunistic, where programmers look for clues in the code

that point to key elements they know must exist. These key elements are then examined in detail to fill in the models.

Guiding the development of the models is the use of hypotheses [von Mayrhauser and Vans 1996b]. As programmers examine the system, they examine features and hypothesize about their purpose. For example, "Variable iMode appears to be a flag controlling display behavior." The programmer then uses systematic or opportunistic searches to verify or reject this hypothesis, switching between the three models as needed. When verification or rejection is achieved, this knowledge is chunked and stored.

2.4.1.4 Boehm-Davis, Fox, and Philips

Boehm-Davis, Fox, and Philips favor the bottom-up approach to comprehension [Boehm-Davis *et al.* 1996]. Like Soloway, they theorize that as programmers gain expertise, they develop a stronger repertoire of plans. Like von Mayrhauser and others, they theorize that programmers segment code into a smaller set of abstractions through an integration process. Through a process of hypothesizing and segmenting, new knowledge is integrated into the current understanding of the problem.

Their research focused on attempting to identify the low-level units of code that programmers identify as part of this process. Experiments conducted using programmers with a range of skills revealed that unit identification ability increased as the skill level increased and that the units identified were sensitive to program structure.

2.4.2 The Need for Tools

von Mayrhauser and Van make a strong argument for tools that support the cognition models defined previously [von Mayrhauser and Vans 1993]. They observe that most tools on the market in 1993 were primarily concerned with supporting only the program model of comprehension. Others were primarily concerned with the top-down domain model (CASE tools). The ability to switch between the different models of comprehension was not commonly supported. While this may seem dated, even as recently as 1997, Favre noted that commercial tool support for understanding large software systems is quite limited [Favre 1997].

Storey, Fracchia, and Müller propose a set of cognitive design elements that should be included in any tool that supports software exploration [Storey *et al.* 1997]. These elements are based on the cognitive models described previously. The design element hierarchy is shown in Figure 2-4.

2.4.2.1 Software Visualization

The general idea behind visualizations of software is that an enhanced or alternate view of the system's code will assist developers in understanding and manipulating it. For the purposes of this work, discussion will be limited to two visualization areas of interest: visualizations of the overall architecture of a system, and visualizations of the a system's code.

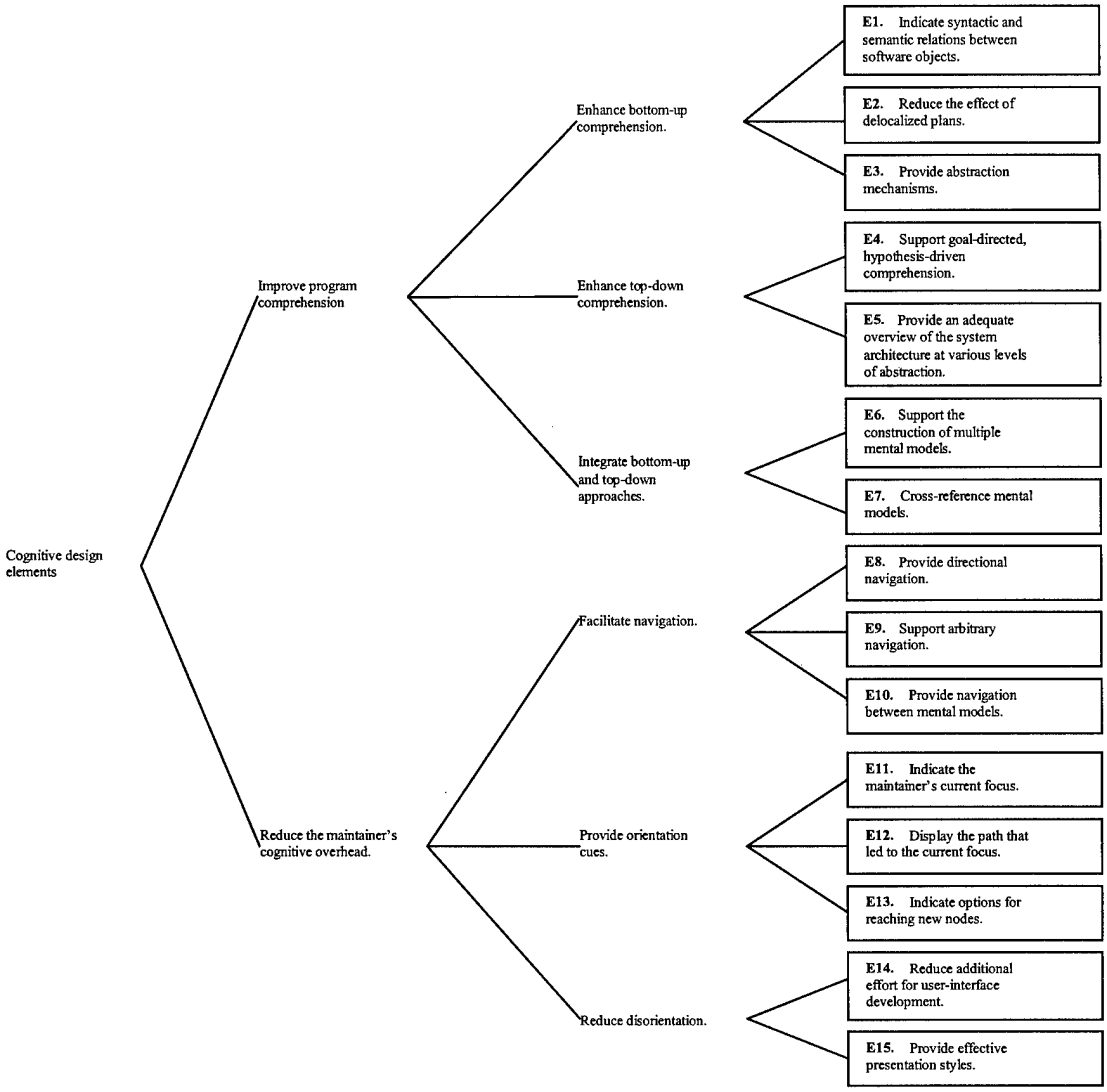


Figure 2-4. Cognitive Design Elements for Software Tools [Storey97]

2.4.2.2 Architecture Visualization

The idea of visualizing the overall architecture of a system is best explained by what Soloway referred to as explanations and mechanisms [Soloway 1986]. In these terms, code is seen as the mechanism for accomplishing a task. One sees the structure of that code, however, as explaining why the code exists. When these structures are combined,

the domain model is completed, and comprehension is gained as to what the system does. The following sections offer some methods that have been developed for visualizing software architecture.

STRUCTURE DIAGRAMS

One of the earliest mechanisms to appear for visualizing the structure of a system was the structure chart. In its basic form, the structure chart shows a hierarchical representation of functional entities in the system. The relationships between these entities was in the form of a procedure call. Later enhancements to structure charts added annotations to the links between functional entities describing data and control being passed.

In the late 1970's, structured analysis and structured design techniques formalized by [DeMarco 1979] and [Yourdon and Constantine 1978] led to the introduction of information flow visualization. In this case, a system is not viewed simply as an organization of modules calling each other, but instead as a set of processes that manipulate information flowing between them. Processes are represented by annotated circles, information flow is represented by annotated arrows between the processes.

OBJECT DIAGRAMS

As object-oriented programming became increasingly popular in the late 1980's, experts began to look for methods by which analysis and design could be extended to use object-oriented techniques. James Rumbaugh et al. [Rumbaugh *et al.* 1991], Grady Booch [Booch 1994], Peter Coad and Edward Yourdon [Coad and Yourdon 1991a, 1991b] developed strategies for analyzing and designing software using objects.

While techniques vary slightly, a common element amongst all the authors was the concept of an object diagram which describes an object-oriented software system. The diagram consists of classes, with connectors between them that describe inheritance, use, and ownership associations. Rumbaugh et al. and Coad-Yourdon, use squares with internal annotations to represent classes; relationships are shown with lines between the classes that have various decorations and annotations to describe their purpose. Booch uses cloud shapes for classes; relationships are shown with lines between the classes where a line with an arrow indicates inheritance, lines without arrows indicate has or use relationships. As with Rumbaugh and Coad-Yourdon, decorations and annotations may be added to the relationships.

Recently, Rumbaugh and Booch worked together to produce the Unified Modeling Language (UML) for Rational Corporation [Rational Corp. 1997]. Object diagrams in UML are essentially a merging of the original diagrams used by Rumbaugh and Booch.

DISTORTION VIEWING

Leung and Apperley point out that one of the problems with viewing large software systems is the physical size of the display monitor that the programmer uses to work with the code [Leung 1994]. If a system consists of many thousand functional elements, or several hundred object classes, then viewing the architecture of the system becomes difficult. Leung and Apperley looked at a set of methods for solving this problem that involve distorting the diagrams such that elements of interest are displayed prominently

and items of peripheral interest are de-emphasized in the display. Of particular interest in the taxonomy they present are perspective wall and fisheye views.

The Perspective Wall as described by Mackinlay [Mackinlay *et al.* 1991] splits the display into three segments arranged horizontally from left to right across the screen. The center segment displays a portion of the diagram being examined in a normal or possibly magnified view. The left and right segments show the peripheral edges of the diagram in an increasingly demagnified view as the segment approaches the left or right edge of the screen. The effect is such that it appears the diagram is "receding" as it approaches the edge of the screen.

Graphical Fisheye Views were presented by Sarkar and Brown [Sarkar and Brown 1994] as an extension of earlier work done by Furnas [Furnas 1986]. Fisheye views get their name from the camera lens that magnifies the center portion of a picture and demagnifies it increasingly toward all edges of the picture. Furnas originally proposed viewing textual information organized in a hierarchical manner using this view concept. Sarkar and Brown extended this to the field of graphical structures, allowing views of software architecture to be seen in clarity at the center of the view area, but reducing it according to a defined mathematical function as elements become farther away from the view's center.

A novel approach to visualization has been developed by Citrin, Santiago and Zorn in a tool called VIPR [Citrin *et al.* 1996]. VIPR represents programs as concentric rings, where the order of the rings implies flow of control. Groups of concentric rings form

functions, and relationships are shown as arrows between them. To allow the display of large systems, or programs with deep nesting, the rings may be zoomed in or out. Since code is displayed as rings, there are few edge crossings, making the diagram stay clean as it is scaled up and down. The authors combine this approach with fisheye views to allow users to move functions in and out of the focus area, zooming in and out as necessary.

2.4.2.3 Code Visualization

Support for Pennington's concept of the program model [Pennington 1987] is supplied by visualizing the code itself. The intent is that by supplying a good visualization to support the program model, that making the connections required to develop an internal situation model will be easier. The following sections describe techniques that are used to visualize code.

TEXT FORMATTING

Possibly one of the oldest methods for visualizing code involves arranging the textual representation so that it conveys meaning to the reader. Indentation of key structures and manipulation of text color and font are the main methods by which this type of visualization is done. Termed "pretty printing" by Ledgard [Ledgard 1975], most modern development environments now include some facility for automatic indentation and coloring of syntactic elements.

In a study conducted by Miara, Musselman, Navarro, and Shneiderman [Miara *et al.* 1983] they found that indentation correlated strongly with comprehension. They tested

novice and expert programmers using a series of programs with indentation levels of 0, 2, 4, or 6 spaces. Results showed significant improvement in comprehension with indentation levels of 2 or 4 spaces, especially among novice programmers. At 6 spaces, however, comprehension fell below that of unindented code.

Tapp and Kazman conducted experiments to determine the effectiveness of color and font in programming tasks [Tapp and Kazman 1994]. They found that in both experiments that the use of differing fonts had no significant effect on the outcome of the programming task as compared to using code with a single font. When color was introduced, however, they found a significant improvement in the speed at which a task could be performed. Subjects of the study liked the use of differing fonts, but preferred the use of color far more.

ABSTRACT CODE VISUALIZATION

An alternate method of constructing the program model is not to view the code itself, but instead look at a graphic representation of what it is doing. Flowcharts were one of the earliest tools created for this purpose. Each line code, or grouping of lines with similar purpose, is represented by a symbol. These symbols include constructs for sequential computation, input/output, selection, etc. Lines connecting the symbols show the flow of control through the software.

Nassi and Shneiderman developed a method of diagramming code similar to that of flowcharting [Nassi and Shneiderman 1973]. In their mechanism, structured programming techniques are enforced by how the diagram itself is constructed. Nassi-

Shneiderman diagrams consist of a tall rectangular block that is segmented from top to bottom. Control flow moves from the top segment through to the bottom. Segments can represent sequential statements, conditional statements, or loop constructs. Segments such as loops wrap around other segments to show containment, and to enforce structure.

Another interesting way to view code was introduced by Eick, Steffen, and Sumner in the AT&T SeeSoft tool [Eick *et al.* 1992, Ball *et al.* 1996]. SeeSoft takes a file and displays the code in it by representing each line of code as a thin line of graphic pixels. The net effect is to make it look as if the programmer were seeing an outline of the code. Additionally, SeeSoft can color each reduced line according to the age of line of code, the number of changes made to it, the author who wrote the code, etc. When this view is combined with a normal non-reduced view of the code, it gives the programmer some contextual clues about what the code's purpose is.

GRASP

In the Graphical Representations of Algorithms, Structures and Processes (GRASP) research project underway at Auburn University, researchers have sought to develop practical visualizations of algorithms, structures, and processes for use in solving real-world software engineering problems [Hendrix *et al.* 1998]. The GRASP software tool developed to support this purpose has been supported by NASA, DoD and NSF.

GRASP takes a hybrid approach to code visualization. It combines indentation, color, and font use with the conceptual advantages provided by abstract views such as flowcharts. This is done primarily through the use of the Control Structure Diagram

(CSD). The CSD overlays program code to show the organization and flow of control within software modules. Symbology for the CSD is similar to that of a flowchart, with symbol additions made to show function and package organization. Figure 2-5 shows an example of an Ada function with the CSD rendered.

Since GRASP is designed to be useful outside the academic setting, the ability to quickly generate the visualization is considered essential. GRASP is able to generate the CSD automatically and with very small time and storage overhead (on the order of 5,000 to 10,000 lines per second). Programmers can keep the CSD active without worrying about it increasing their workload or slowing down the environment.

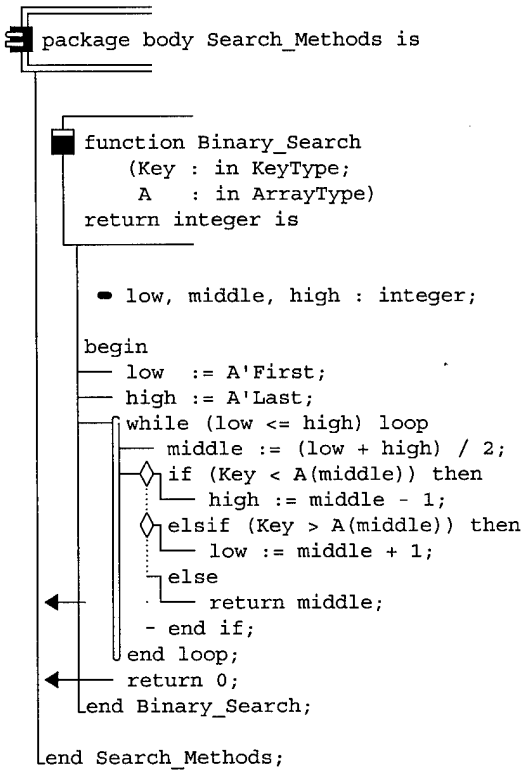


Figure 2-5. CSD Example

GRASP appears to provide direct support to Pennington's bottom-up method of comprehension. It supplies a pre-made abstraction of the program model right next to the code being examined. In the near future it is expected that further support of this conceptual model will be added when structural folding is introduced. Folding allows the user to close ("fold") code that is not of immediate interest to the programmer. For example, a module could be folded, or an if statement could be folded, or the contents of a while loop, etc. In essence, as programmers chunk code into the situation model and program models, they can fold them in the development environment.

Another view of this approach with folding is to consider in terms of a top-down approach. Here, programmers start with the entire set of code folded. As they take opportunistic searches through the code, they unfold only those elements of interest.

2.5 DATA COLLECTION IN USABILITY STUDIES

Discovering which elements of a user interface are effective, and which are not has been an ongoing area of research. As with comprehension studies, techniques usually involve a combination of direct observation and video taping of subjects. Research into automated data collection during studies of usability, however, is much further along than with comprehension studies. Work in this area has progressed from simple instrumentation and logging to advanced remote agents that selectively filter user input data. This work is examined in the following sections.

2.5.1 Instrumentation and Logging

As user interfaces have become more sophisticated, methods for evaluating them have improved. In 1991, Siochi and Ehrich used a technique in which a user interface management system (UIMS) was instrumented so as to write transcripts of user sessions [Siochi and Ehrich 1991]. This technique had been done before, as they noted in previous work by Neal and Simons where two computers were connected so that one could record keystrokes from the other [Neal and Simons 1983]. What Siochi and Ehrich suggested, however, was that simply recording data is not sufficient. They equated this to merely another form of videotaping.

Their experiment introduced the idea of automatically analyzing the transcript for useful data. Specifically, they searched for repetitive patterns of user actions, theorizing that this would indicate a defect in the UIMS design. Seeing great potential in this technique, they recommended further work be conducted in broadening automated analysis and that transcript entries should be enhanced with timestamp information.

A very large usability study was conducted at the University of Sydney using logging [Kay and Thomas 1995]. In this study, an editing program called "sam" was instrumented to write a log file of various commands and user activities. This log file was written to a hidden file in the user's directory and transferred to a central location automatically each night. A program was then used to analyze the logs for each user.

Kay and Thomas found this technique to work quite well. They studied 2,273 undergraduate students from 1991 to 1993. The method of writing files had a low impact

on the system and was resistant to system crashes and network failures. They were able to successfully use the data to build up long-term trends in how programmers learned to use the editor.

2.5.2 Distributed Collection and Expectation Agents

As network technology has improved and more locations are tied together using high-speed networks, the idea of using this technology for distributed collection has been investigated. In 1995, Bates reported on work he had conducted in debugging technology [Bates 1995]. His method of data collection relied upon the generation of event information that was generated as the user worked, and analyzed in real-time by behavior recognition systems.

Hilbert and Redmiles take the idea of event generation and analysis in real-time a step further [Hilbert and Redmiles 1998]. They note that in complex systems the data burden of recording events such as mouse movements and keystrokes can be tremendous for large numbers of users. The volume of this data would quickly overload a network if it were being sent across as it was being created.

Their solution to this problem was to create expectation agents. An expectation agent can be thought of as the developer's view on how the application being monitored will be used. When the agent detects that there is a mismatch between the use of the application and developer expectations, it sends data to the developer for analysis. Called event-driven expectation monitoring (EDEM), this significantly reduces the network burden.

CHAPTER 3. THE DATA COLLECTION PROBLEM

3.1 INTRODUCTION

Based on the available literature, as discussed in the previous section, there appears to be a true disconnect between measures of software attributes and measures of program comprehension. The goal is to establish a relationship between a software measure such as complexity with comprehension. However, before that goal can be achieved, a first step must be taken: there must exist methods and tools by which this information can be collected, represented accurately, stored, and analyzed. The focus of this dissertation has been to research and solve this collection, representation, storage, and analysis problem.

This chapter discusses the problems that exist in collecting software measures, and proposes a set of solutions that act as the requirements for the collection framework. Chapter Four discusses how the framework's design and implementation meets these requirements.

3.2 CORRELATION OF COMPREHENSION AND SOFTWARE MEASURES

3.2.1 Problem Description

Many of the studies conducted with respect to program comprehension concentrate on actions taken by the programmers as they look at code. While most consider the experience and skill levels of the subjects, few consider the complexity of the software

the comprehension measurements are being taken against. von Mayrhauser and Vans have concentrated on programmers who deal with large scale systems, but even their studies simply quote LOC numbers when discussing the complexity of the code. Since LOC is regarded as a poor measure of software size and complexity, this introduces another variable into comprehension studies that has largely been ignored.

3.2.2 Proposed Solution

What is required is that not only must measurements of comprehension be taken, but the software against which these measurements are being taken must also be evaluated. In order for this to happen, there must exist an underlying framework for collecting this information and analyzing it. This framework should be capable of collecting the following types of software measures:

- *Size Measures.* The system should be capable of collecting basic measures such as LOC counts, function point counts, and Halstead's Software Science measures.
- *Complexity Measures.* The system should be capable of collecting McCabe's cyclomatic complexity number per module, McCabe's design complexity, and McQuaid's CPG.
- *Object-Oriented Measures.* In order to account for studies where object-oriented code is being used, the system should be capable of collecting Chidamber and Kemerer's suite of measures.
- *Other Measures.* Other potential measures that may need to be collected include those proposed as coupling and cohesion measures.

3.3 DATA COLLECTION

3.3.1 Problem Description

One of the problems with understanding software comprehension is that the programmer must be observed in one form or another. This presents a problem since the act of observing can change the behavior of the subject. Techniques like "thinking aloud" where the programmer vocalizes their thought processes suffer from similar problems, requiring researchers to set up extra control groups to ensure the observation process itself is not having an effect on the experiment.

Direct observation is also a problem when large numbers of subjects are involved. Extensive work is being conducted at Auburn University designed to determine the usefulness of visualizations such as the control structure diagram (CSD), complexity profile graph (CPG), and control structure folding. In order to determine how useful programmers find these tools, it is desirable to monitor their use patterns. It is not feasible to do this by observation due to the large number of subjects, often 100 or more, involved in the studies. In addition, some studies involve allowing the subjects to work in diverse environments where direct observation is not possible.

In terms of industry use, the personal software process (PSP) developed by Watts Humphrey requires extensive time measurements in the different development phases [Humphrey 1997]. This requires use of "diary style" logging of timing data. This type of collection has been shown to be inaccurate by Dewayne Perry at Bell Laboratories, who resorted to having personnel physically observe programmers in order to get accurate

information [Perry *et al.* 1994]. Basili also noted that excessive overhead for the programmer introduces inaccuracies into data and recommended automating as much as possible [Basili and Weiss 1984]. Since Humphrey's computations depend on accurate data for predicting productivity, the more accurate this information is, the better the resulting prediction.

3.3.2 Proposed Solution

Since direct observation, thinking aloud, and diaries have side effects and limitations, the best alternative is automatic measurement through the development environment itself. To this purpose, automatic measurements of the following need to be made:

- *Session Statistics.* The system must record when a user started a session and when a user finished it. To support PSP, the system will allow the user to specify what phase they are in (analysis, design, coding, testing) and the system will record the amount of time spent in that phase. It will allow the user to record interrupts such as phone calls and personal breaks.
- *CSD Usage.* The system must count CSD generations, both attempted and successful. It will measure how long the CSD was active as a percentage of the session time of the user. Unsuccessful generations will have the error line recorded.
- *Compiler Usage.* The system must count the number of compiles, both attempted and successful. It will record errors and their line numbers.

- *Folding Usage.* The system must record when a folding or unfolding action takes place. It will record the number of lines folded or unfolded.
- *File Usage.* The system must record file openings and closings, as well as recording the amount of time the file was being actively edited.
- *Line-Oriented Measures.* The system must record when a line in a file was displayed, when it was added, when it was changed, and when it was deleted.
- *User-defined Events.* The framework must allow users to manually inject events containing arbitrary string or numeric data. This will allow PSP users to record logical errors that cannot be automatically tracked by the environment, and researchers to record comprehension test scores.

This discussion should not imply that automatic recording of measurements is necessarily a substitute for direct observation. Studies needing to know the motivation behind why a programmer took a certain action will have to ask the subject directly. Automatic measurement can be used in conjunction with these experiments to assist in analysis of the data and possibly help support conclusions.

3.4 DATA REPRESENTATION

Given that data must be collected as described previously for both software and comprehension measures, the next problem involves the correct representation of that information. The representation problem can be divided into three parts: how software measures are represented, how comprehension measures are represented, and how both of these measures can be represented in a single framework.

3.4.1 Problem: Software Measure Representation

One of the problems found by Fenton, Kitchenham, and Pfleeger in their examinations of the various software measures is that researchers often do not adhere to sound measurement theory [Fenton and Pfleeger 1997, Kitchenham *et al.* 1995]. In particular, they often find the researchers apply inappropriate statistical analysis to measures because they fail to understand the limitation of the measure's scale type.

Measurement theory defines several scale types [Fenton and Pfleeger 1997]:

- *Nominal.* The nominal scale is a simple categorization of the measure. There is no order of magnitude implied for the categories. For example, if one was measuring defects in a compiler, a nominal scale could be used to indicate where the defect occurred: preprocessor, lexical analyzer, code generator, optimizer, etc.
- *Ordinal.* This scale categorizes the measure, but applies an ordering to the categories. This is a simple ranking and operations like addition, subtraction, etc. will have no meaning. An example of an ordinal scale might be the severity of a defect: low priority, medium priority, high priority. The ordering "less important than" applies to this order.
- *Interval.* In this type of scale, there is an order, and the size of the interval between classifications is known. Addition and subtraction between measures may be performed, but not multiplication and addition. As Fenton notes, there are few cases in software engineering where this type of scale is employed. An

example of this scale is temperature: it may be 30 degrees Celsius in New York and 20 degrees Celsius in London. It can be said that New York is 10 degrees warmer than London, but not that Washington is 50% warmer than London.

- *Ratio.* The ratio scale has order, defined intervals between classifications, and a defined starting point representing the lack of a measured attribute (zero). All arithmetic operations are allowed between measures. An example of this type of scale would be the size of a software system measured in LOC. If system A has 100 LOC, and system B has 150, it can be said that B has 50% more code than A.
- *Absolute.* This scale is simply a count of the attribute. In order for this to be valid, there must be only one way to count the attribute. All mathematical operations may be applied. An example of this type of scale might be the number of defects in a system. In this case, it can be said that system A with 10 defects has twice as many defects as system B with 5. Note that size of a system measured in LOC is not absolute since size can be measured several ways.

Scale is important in order to avoid making statistical mistakes with information. For example, it would be incorrect to say that high priority defect A is three times as important as low priority defect B. Both are measured on an ordinal, not ratio scale.

A more common example would be the use of the cyclomatic number as a mapping to to integers that defines a software unit's complexity. If one accepts that this mapping applies to the ordinal scale, then it would be incorrect to say that unit A with a cyclomatic number of 8 is twice as complex as unit B with cyclomatic number 4. Further, Fenton

argues it cannot even be accepted that this is a correct mapping to the ordinal scale since the "more complex than" relation is not a strict weak order. Basically, because of the vague definition of complexity, given two programs with differing structure but the same cyclomatic number, it is unlikely that all programmers would agree that one was more complex than the other. Given these issues, it is important that scale selection and measurement manipulation be carefully controlled.

3.4.2 Problem: Comprehension Measure Representation

Software measures are generally static in nature. For example, to count the LOC of a program as a measure of size, this would be done after the program has been written, not during its construction. Similarly, if to measure the complexity of a program, one would do this as a "snapshot" of the software at a given point in time.

Software comprehension measurements, however, often have a dynamic aspect. While it is normal to have a comprehension exam after an experiment, it is very common to have measurements being taken during the experiment. For example, using the think aloud technique, programmers describe their actions as they examine software. Observers record these comments for later analysis. The order in which the comments are made is important and must be preserved. Because of this, time becomes a factor in the unit selection for measurements.

3.4.3 Problem: Measure Integration

If it is necessary to record both software and comprehension measures, then selection of scale and units can be a problem. The time order of the comprehension measurements must be preserved and software measures must be taken so they can be correlated with the comprehension measurements. For example, it may be noted that one hour into the experiment, a programmer started a compile that ended with five errors. At one hour thirty minutes, the programmer starts a compile that ends with two errors. At two hours, the programmer starts a compile that ends with no errors. Although one may want to know the cyclomatic number of the programmer's code, until a successful compile is achieved it cannot be collected automatically. It is possible, however, to take comprehension measurements. A collection framework must be able to represent both types of measurements so they can be correlated.

3.4.4 Proposed Solution

In order to solve the above three main problems, the idea of recording information in summary form is discarded. Instead, information is recorded as a series of events. Each event has an inherent time associated with it. After events are recorded and the session is complete, the data can then be analyzed. Since time information will be preserved, it is possible to recreate the session by simply looking at the sequence of the events.

Additionally, by storing detailed event information, the problems with retaining the proper data representation are minimized. The framework cannot enforce a scale per se, but it can offer the data in such a manner as to encourage the proper use of the scale.

Integration with software measurement data is not an issue. Software measurements are considered events and stored as such. The time component can be used to determine what the most current measurement of the software is, and to look at the change in the characteristics of the software over time.

3.5 DATA STORAGE AND ANALYSIS

3.5.1 Problem Description

There is a compelling interest for commercial users of software measures to track information over a period of time [Grady and Caswell 1987, Ross 1987]. This history provides managers and programmers the ability to predict how long a future project may take based on historical performance. It provides feedback for companies attempting to improve their development processes. For researchers, it may take several studies to validate a theory pertaining to software measurement, comprehension, or both.

3.5.2 Proposed Solution

To meet the needs of industry and research, the measures need to be stored in a database. This requires the design of a database system that can represent information as discussed in Section 3.4. In order to maintain maximum flexibility, the system cannot tie itself to a single vendor's database, but should be able to work with any database capable of accepting ANSI SQL 1992 data manipulation language and data definition language commands.

Further, the system must be able to save information offline from the database when necessary. It can be anticipated that some measurements will be taken on machines that are not tied directly to the database management system, and so these measurements will need to be placed in a file. This file can then later be imported into a database for analysis.

CHAPTER 4. THE DATA COLLECTION FRAMEWORK

4.1 INTRODUCTION

Chapter 3 defined the problems associated with collecting measurement information properly and proposed general solutions to those problems through the creation of a data collection framework. In this chapter, the mathematical basis of the collection framework is discussed, followed by a detailed examination of its design and implementation. A sample analysis package developed to assist in using the framework is also presented. Chapter 5 discusses how this framework was tested by integration with an actual development environment.

4.2 THE FRAMEWORK CONCEPTUAL MODEL

The goal of the framework is to develop a representation for user actions and software state that is a useful basis for analysis. There are a finite number of possible user actions within a development environment. These actions include activity such as opening a file, moving the cursor, performing a compile, etc. The finite set S_a is defined to contain elements representing these actions. The set S_b is defined to be the set of all possible states of the software being operated upon. Since the possibility of external modification is allowed, this set of states is infinite.

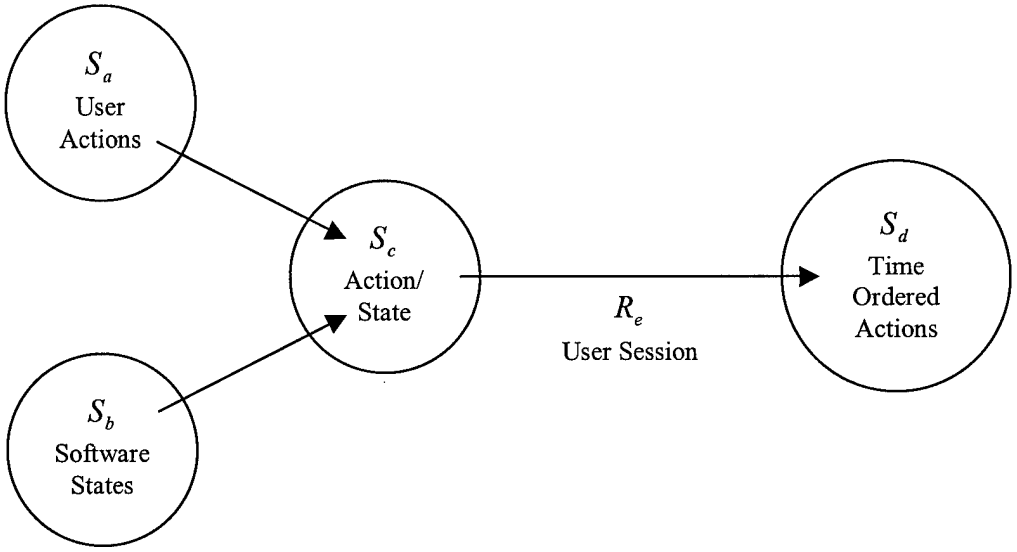


Figure 4-1. Conceptual Model for the Framework

In order to provide correlation between these two types of data, the cartesian product of the sets is taken: $S_c = S_a \times S_b$. Then S_c contains tuples of the form $< x, y >$ where $x \in S_a$, and $y \in S_b$ where y is a set of tuples that describe the state of the software in terms of its size and structure.

The final step is to create a time-ordered set of action-state tuples. The set S_d is defined to be a set of tuples of the form $< t, z >$ where $t \in N$ (time in natural numbered units) and $z \in S_c$. The user session then becomes a relation R_e between S_c and S_d such that elements are selected from S_c for inclusion into S_d as part of the editing process. The purpose of the framework is to capture R_e so that the set S_d can be constructed as needed for analysis. The model is shown in Figure 4-1.

Determining the amount of editing time a file has had during a session is an an example of the use of this type of correlation. We take pairs of tuples $< x_{ij}, y_{ij} >$ from S_d where $x_{ij} \in$ (file activation i on file j) and $y_{ij} \in$ (corresponding file deactivation i on file j). Then the total file activity is:

$$T_j = \sum_{i=1}^{n} (y_{ij}.t - x_{ij}.t)$$

Another example is the computation of program development times in PSP. In this case, one computes LOC per minute as:

$$Rate = \frac{(LOC_{start} - LOC_{end})}{n}$$

But this shows little about variation in rates over the period of development. Using the information in the set S_d the mean LOC can be computed as:

$$\overline{X} = \frac{\sum_{i=1}^{n-1} (y_{i+1}.A - y_i.A)}{n}$$ where $y_i.A$ is the LOC at interval i.

Standard deviation is then computed as:

$$s = \sqrt{\frac{\sum_{i=1}^{n-1} ((y_{i+1}.A - y_i.A) - \overline{X})^2}{n}}$$

If development rates in terms of LOC vary significantly over the period of measurement, then the standard deviation will be large. This will signal to the person performing the analysis that growth of the code was not linear, and this data may not be useful for predicting the growth of future projects which exhibit linear growth.

The following sections discuss how the mathematical model was translated into computer software via object-oriented design and programming techniques.

4.3 DEVELOPMENT METHODOLOGY

There are two main alternatives for software development: structured and object-oriented. While there is a justification for using structured techniques because of the data manipulation involved, which lends itself to data-flow diagramming, the method selected was object-oriented analysis, design, and programming for the following reason. The development community is almost universally accepting object-oriented techniques as the proper methodology, so it is probable that environments to which the framework will link will be object-oriented.

The following sections outline the basics of the object-oriented development process, and briefly discuss how these impacted decisions made in creating the framework.

4.3.1 Object-Oriented Analysis (OOA)

During the analysis phase of development, the problem statement is refined so that solutions can be designed and implemented. In OOA, this is done by examining the problem and identifying objects and their relationships with each other. An object can be thought of as something that has state, defined behavior, and identity [Booch 1994, Rumbaugh *et al.* 1991]. For example, a car might be considered to be an object in a traffic simulation. It has a state: speed, fuel level, color, etc. It has behavior: accelerate,

turn, brake, etc. It has an identity in that this car can be distinguished from other cars by its license plate.

In the terms of a framework, the definition of an object can be less clear since it is not a real world object. As will be presented later in the discussion of the architecture, the defined objects are conceptual. For example, the file manager is an object that has state: the file being manipulated. It has behavior: read events from files, write events to files. It has identity in that each instantiation of the framework has one unique file manager it uses.

The output of OOA is a set of class diagrams that show the relationships between objects, a set of interaction diagrams that show how objects interact with each other, and state-transition diagrams that show how objects change as they operate.

4.3.2 Object-Oriented Design (OOD)

During design the set of objects is organized into systems and subsystems. Each object is examined and refined to determine its specific state factors, operations, and relationships with other objects. The class structure defined in OOA is optimized and expanded as necessary to give a robust definition of each object, class of objects, system utilization, and system utilities required.

4.3.3 Object-Oriented Programming (OOP)

The act of translating the design into a program which uses objects and operates on them as specified is object-oriented programming. It is not necessary to have a language that

explicitly supports OOA/OOD to implement the constructs of inheritance and polymorphism that OOA/OOD uses when defining object classes. However, languages such as C++ and Java have explicit support for these constructs and this tends to eliminate a lot of extra code. As will be discussed later, the framework uses the programming language Java because of the ease of implementing the object-oriented design, and to maximize the portability across platforms.

4.4 ARCHITECTURE

The first task in creating a software implementation of the conceptual model is deciding the best manner to represent the sets S_a, S_b, S_c and S_d. Further, it is necessary to more precisely define the contents of an element y in set S_b. Finally, a mechanism for capturing the selections of elements from S_c into S_d must be designed.

Some basic organizational decisions have to be made as part of developing a representation, since there are two major ways to implement the model. In the first, code is written into each development environment giving them the ability to write data to some media, and then separate applications are written that read the data. This has a severe disadvantage in that a lot of code will have to be written into an application to support the handling of the data. This same code will be repeated in the analysis application which has to examine it. Additionally, if new environments are developed, the same code has to be embedded again. This unnecessarily complicates the software.

The second and more practical way to implement the solutions is via a loosely coupled framework of objects. "Framework" in this sense refers to a set of objects that work

together through a defined set of dependencies and interaction rules. A user of the framework inserts their objects into it in such a manner as to obey the dependencies and interactions. In this section, the mapping of the conceptual model to an object-oriented framework is described, and the rules for the framework are specified.

4.4.1 Set Representation

In order to allow the framework to work with a variety of applications, it is essential that the representation selected for the sets not be rigid. It is understood that as the framework is used for new applications, or old applications gain new features, that the contents of the sets will necessarily change. Therefore, what will be established here is a pattern for the sets, rather than a unique enumeration of their content.

The first set examined is S_a, which is defined as the set of all possible user actions in a given application. Similar to the notation used by [Bates 1995], a tuple of this set can be seen as the following general template:

$$(\text{action\_name}, a_1, a_2, ..., a_n)$$

Where action_name is the action being taken, and $a_1, a_2, ..., a_n$ are the attributes associated with that action. Examples of specific actions would then look like:

$$(\text{file\_open}, file\_name, window\_handle)$$

$$(\text{enter\_line}, window\_handle, line\_number)$$

$$(\text{CSD\_generate}, window\_handle)$$

The set S_b holds all possible states of the software being manipulated through the development application. It is defined in a similar manner as S_a:

$$(\text{sw\_name}, b_1, b_2, ..., b_n)$$

Where sw_name is a unique identifier for the software element, and $b_1, b_2, ..., b_n$ are attributes that describe the software's state. State tuples would then look like the following examples:

$$(\text{window\_handle}, physical\_LOC, logical\_LOC, cyclomatic\_complexity)$$

$$(\text{window\_handle}, classes, methods, associations)$$

$$(\text{file\_name}, size\_in\_bytes)$$

Note that sw_name is unique to the application. Some applications may identify software elements by filenames, others may use operating system file descriptors, or in the case of pcGRASP, unique integers called "handles."

As noted in the discussion on the conceptual model, S_c is the cartesian product of S_a and S_b which gives us the set of all possible actions on all possible states. Since tuples are of the form $< x, y >$, then our representation becomes:

$$(\text{action\_name}, a_1, a_2, ..., a_i, \text{sw\_name}, b_1, b_2, ..., b_j)$$

Possible state/action pairs in this set might be:

$$(\text{file\_open}, file\_name, \text{window\_handle}, physical\_LOC, logical\_LOC, complexity)$$

$$(\text{file\_compile}, errors, \text{file\_name}, size\_in\_bytes)$$

As the user works in the development environment, different actions will be taken, and the state of the software will change. The work occurs in a specific order, and this order is represented by the set S_d which adds the element of time. So the representation of this set is then:

$$(t, \text{action\_name}, a_1, a_2, ..., a_i, \text{sw\_name}, b_1, b_2, ..., b_j)$$

Where t is the time at which the action took place. Since this representation combines the action/state with a time, it is referred to as an "event." The purpose of the framework is to collect and store events as represented above. The following sections describe in greater detail how this representation is converted to an object-oriented design.

4.4.2 Overall Architecture

The architecture of any system of software which uses the framework will have three major components, as shown in Figure 4-2. The development environment is created by

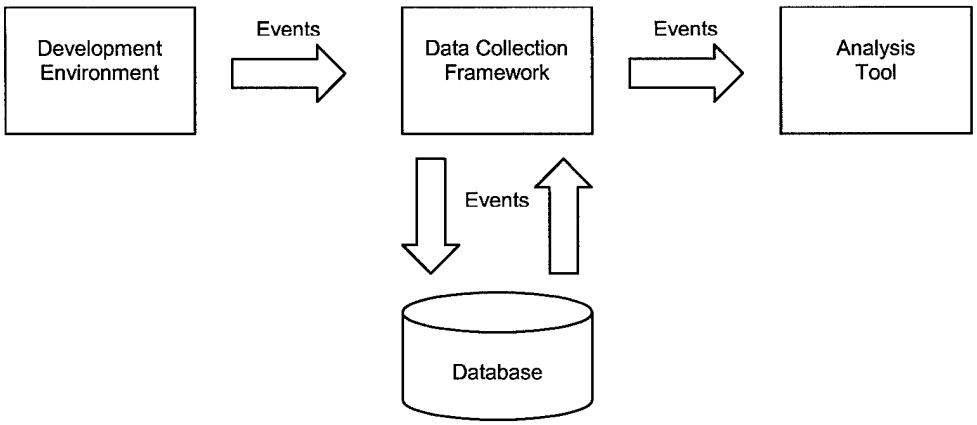


Figure 4-2. Framework Architecture

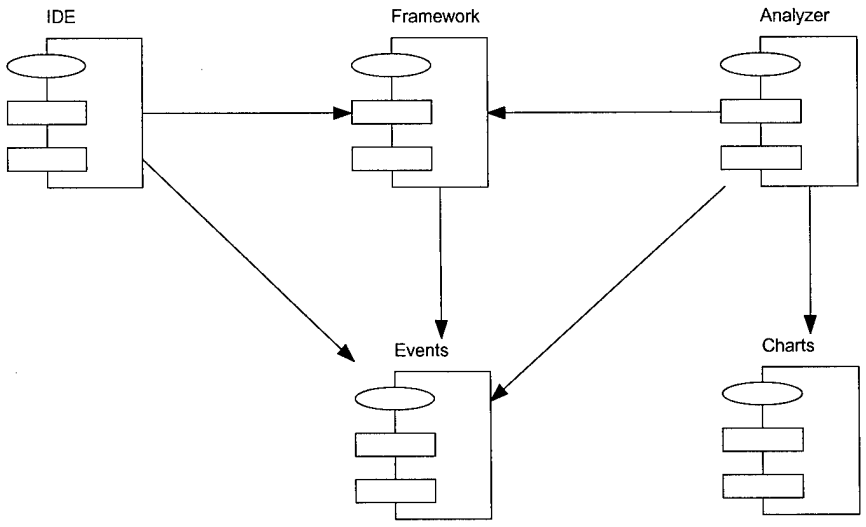


Figure 4-3. Software Package Dependencies in the Framework

the user of the framework. It communicates with the components of the data collection framework by passing events for storage. The data collection framework organizes and stores these events to files on disk. The analysis tool uses the framework to retrieve and manipulate events.

Figure 4-3 shows the packages and dependencies in the system. The development environment, marked IDE (a common industry acronym for integrated development environment) in the figure, uses utilities and event definitions in the framework and events packages. The events package simply defines all the possible events that can be recorded. The IDE creates instances of these events, such as a user compiling a file, and passes it to an interface component in the framework package for storage. The analyzer package uses the framework package to get events from the database, and uses the events package to allow manipulation of the events.

4.4.3 The Event Hierarchy

Information regarding what is going on within the IDE is stored as events. As per the model discussed earlier, an event is a tuple from S_d which describes a specific action taken on software in a specific state, at a specific point in time. In that set, each tuple is represented as:

$$(t, \text{action\_name}, a_1, a_2, ..., a_i, \text{name}, b_1, b_2, ..., b_j)$$

Each event is considered to be an object which must be stored by the framework. Therefore, an appropriate class is defined for each possible event. The representation above is mapped to a class structure as shown in Figure 4-4. The event is only triggered when a user takes an action, therefore the event is named after that action. The event contains a matching attribute for every attribute of a tuple in S_d, plus an integer representing the time at which the event occurred.

From a mathematical perspective, this conversion to object classes preserves all information and is correct. From an implementation perspective, however, it is too inefficient to actually store data in this manner. First, there is a large set of actions that will not change the state of the software. For example, moving the cursor, opening a file, closing a file, saving a file, etc. These actions do not actually alter the software being manipulated in the environment. Therefore, if the user moves the cursor repeatedly without changing the underlying software, then state data is repeated needlessly.

class action_name:
- **time** t
- **attribute** a_1
- **attribute** a_2
- ...
- **attribute** a_i
- **attribute** sw_name
- **attribute** b_1
- **attribute** b_2
- ...
- **attribute** b_j

(a)

class file_open:
- **Integer** time
- **String** file_name
- **Handle** sw_name
- **Integer** logical_LOC
- **Integer** physical_LOC
- **Integer** complexity

(b)

Figure 4-4. Generic Class from Event (a) and Example (b)

Another problem is the difficulty in assessing the state of the software. Measures of software state such as computing logical lines of code, determining cyclomatic complexity, or calculating the complexity profile require a complete parse of the software. This can be very time consuming, and would probably interfere with the programmer's task if this were done at every keystroke.

To solve this, the representation of events in S_d is changed in the conversion to object classes such that state information is preserved, but repeated data is reduced. This is done by treating the generation of software state data as an action event. The representation of a tuple then becomes:

$$(t, \text{action\_name}, a_1, a_2, ..., a_n)$$

Action Events	State Data
(1,Compute_State,0,100,30,5)	(PhysLOC = 100, LogLOC = 30, Ccmp = 5)
(5,Enter_Line,0,100)	(PhysLOC = 100, LogLOC = 30, Comp = 5)
(6,Enter_Line,0,101)	(PhysLOC = 100, LogLOC = 30, Comp = 5)
(7,Insert_Line,0,101)	(PhysLOC = 100, LogLOC = 30, Comp = 5)
(8,Compute_State,101,30,5)	(PhysLOC = 101, LogLOC = 30, Comp = 5)
(10, Close_File,0)	(PhysLOC = 101, LogLOC = 30, Comp = 5)

Figure 4-5. Example of Modified State Representation

Where action_name can now include a set of actions that cause the associated attributes to be loaded with state information. Examining Figure 4-5 one can see that this new representation does not result in information loss. As the user performs various actions, the last state computation event's data is used to fill in knowledge of the software state. So long as a state computation event occurs each time the software changes, the information will be current. Note that state data for an event is defined to be the state of the software at the time the action is taken, and should not be considered the state of the software after the action is taken.

Using this representation also allows the framework to store estimates of the software state. That is, if computation of the software state is expensive in terms of time, then the application performs fewer state computations. The framework simply uses the last computation recorded, even though it may not be entirely correct due to user changes.

In the following sections, the object classes are further refined and mapped to object classes.

4.4.3.1 The Root Class: CFEvent

The process of examining possible actions and software states for representation reveals that many share common attributes. By examining these common attributes, an inheritence structure can be established. For example, the following tuples both share a common attribute, *window_handle*:

(*t*, file_open, *file_name*, *window_handle*)

(*t*, file_close, *window_handle*)

This commonality, if repeated among many tuples would suggest the presence of a more general abstract category of window or file event. Repeated examination of action events does tend to reveal these kinds of relationships, and an inheritance hierarchy was constructed from them.

The class CFEvent is the root class of the inheritance structure. Figure 4-6 shows this class and its immediate child classes. It records the time the event occurred, a unique

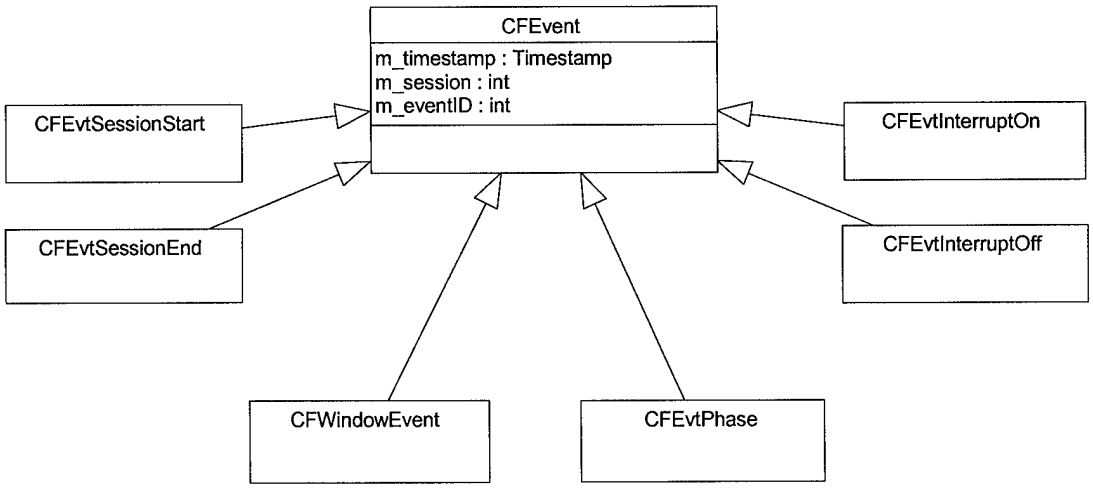


Figure 4-6. Root and Session Events

session identifier that the event occurred in, and a unique identifier for this event. Subclasses add to this information depending on what kind of action they are recording. Thus, the CFEvent class can be seen as a tuple of the following form in S_d:

$(t, \text{CFEvent}, session, eventID)$

Where *session* is an integer that uniquely identifies this session from all other user sessions. As discussed earlier, the user's session acts as a relation R_e from S_c to S_d, this identifier allows the framework to store several of these relations in one dataset while still being able to distinguish between them. It can also be thought of as a way to distinguish one set S_d from another in a dataset consisting of multiple instances of these sets.

Within the framework, all events are handled using this base class to the maximum extent possible. For example, if the analysis tool requests a set of events matching certain criteria, the set will always be returned as a collection of CFEvent objects. The analysis tool can then look at each event as necessary to discover what kind of specialization of CFEvent it is.

4.4.3.2 Session Events

The child classes of CFEvent are known as session events. These events record information that is unique to a specific session. A major abstract class of these events, CFWindowEvent, operates on a specific window in the environment, and forms its own subtree in the hierarchy. The other events which can be instantiated operate outside the

bounds of a specific window or file. Events which can be instantiated always have the prefix "CFEvt" in their name.

There are three main types of session events:

- *Session Control.* The CFEvtSessionStart and CFEvtSessionEnd events put boundaries on when the session began and terminated. This allows an analysis tool to watch for these events and by taking the difference in their timestamp values, can learn how long the session lasted.
- *Personal Software Process (PSP).* The CFEvtPhase event records what phase of development the user has entered. CFEvtInterruptOn and CFEvtInterruptOff allow the programmer to signal when they have been interrupted from work by phone calls, personal breaks, etc. These events allow an analysis tool to determine how long was spent in any given phase of development.
- *Window Events.* This is a class of events that occur within a single session, but are applicable to a specific file or window in the environment. Note that a "window" does not imply a source file, it may be a window containing a structure diagram, symbol browser, debugger, etc.

4.4.3.3 Window Events

The most common type of event is the window event. This event will record a specific action taken by a user, and associate that action with a window in the development environment. As noted previously, a "window" does not always mean a source file. The idea of this class of events is not to put a strict interpretation on what kind of content the

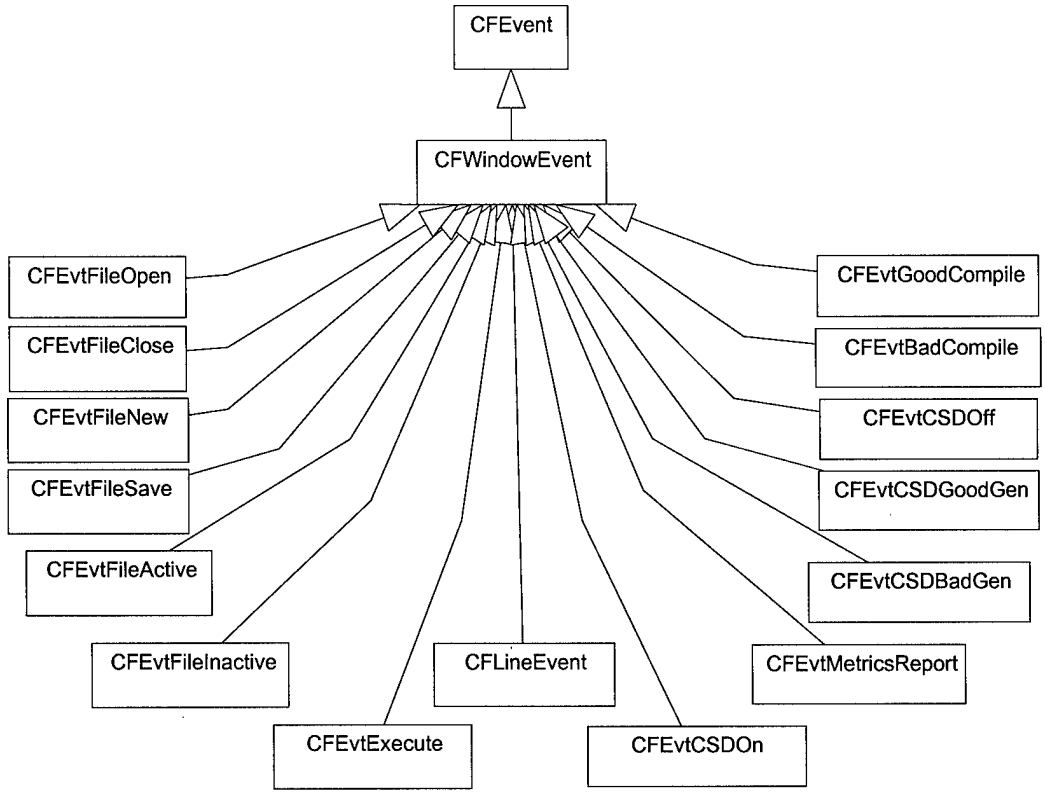


Figure 4-7. Window Events

window may contain. By not assuming it is a source file, one can add events that inherit from this class without unduly deepening the inheritance tree and complicating the design.

Figure 4-7 shows the current set of window events. Currently, they are all oriented toward a source file being in the window. If something other than a source file were there, new events oriented toward that content would be added instead.

The window events can be organized into several major groups:

– *File I/O.* CFEvtFileOpen, CFEvtFileClose, CFEvtFileNew, and CFEvtFileSave all indicate the user took actions which involved reading or writing data from

permanent storage. When a file is created using CFEvtFileNew it does not have a filename. The final filename for the window will be assigned when a CFEvtFileSave is performed. Note that CFEvtFileOpen and CFEvtFileClose have an associated filename.

- *File Usage*. The CFEvtFileActive and CFEvtFileInactive events are used to record which window was active at any given time, if any. When a window is highlighted in the user environment, it becomes active and a CFEvtFileActive event is recorded. When another window or application is highlighted, a CFEvtFileInactive event is recorded. By summing the active periods, an analysis tool can determine how long a user was looking at a specific window in the environment.
- *Compiles*. CFEvtGoodCompile indicates the user compiled a source file in this window, and the compile was successful. CFEvtBadCompile indicates an unsuccessful compile occurred, and stores the number of errors that occurred.
- *Execution*. CFEvtExecute is used to record each time a user executed a specific file.
- Metrics. CFEvtMetricsReport records a logical and physical line count of items in the file, it can also store other file-oriented metrics such as cyclomatic complexity.
- *CSD Events*. This group of events is specific to the GRASP project. CFEvtCSDGoodGen records that the programmer manually generated a CSD

successfully. CFEvtCSDBadGen indicates the CSD generation was not successful, and records the text of the error message. CFEvtCSDOn indicates the environment generated a CSD automatically. CFEvtCSDOff indicates the programmer removed the CSD diagram from the window.

- *Line Events*. This class of events assumes a source file is in the window, and has specific events to record operations on the lines of the file.

4.4.3.4 Line Events

Line events are a subclass of window events. Line events assume there is a source file in the window, and they record a fine-grained level of detail about the operations on the lines of code. Figure 4-8 shows the current set of line events. CFEvtEnterLine indicates

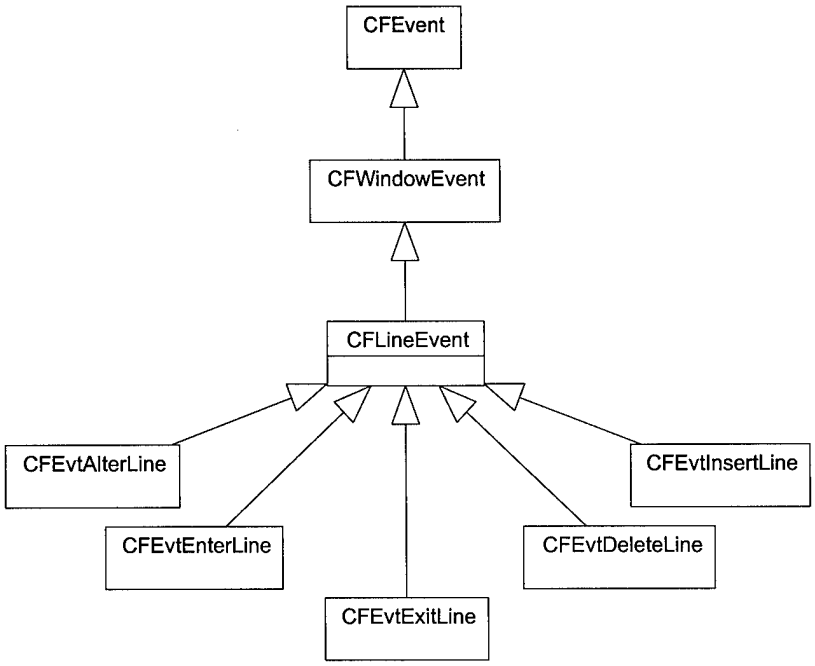


Figure 4-8. Line Events

that a line has been entered by the user placing the edit cursor on it. CFEvtAlterLine indicates that the line was changed by the user through the modification of its text. CFEvtInsertLine indicates that one or more lines were inserted before the current line. CFEvtDeleteLine indicates that one or more lines were deleted, starting with the current line.

By examining these events, an analysis tool can determine not just what file was most active (via the window events described earlier), but what specific part of that file was most active in terms of user activity. This can help managers determine what areas of code may be absorbing most of a team's time. It can be used by researchers to help correlate fine-grained complexity metrics with actual editing time by programmers.

4.4.3.5 Expanding the Hierarchy

It is anticipated that different environments will have unique events they want to capture. The intent of the framework is not to create a rigid set of events that cannot be changed. Instead, it is expected that more events will be added.

If an event in the framework does not precisely meet the needs of the application, then a similar event should be subclassed and overridden as necessary. Doing this will ensure that the core components still work correctly, and that the framework will continue to work with other applications that expect the original set of events.

If there is no similar event, then the event should be added to the tree at the appropriate level of specialization underneath CFEvent.

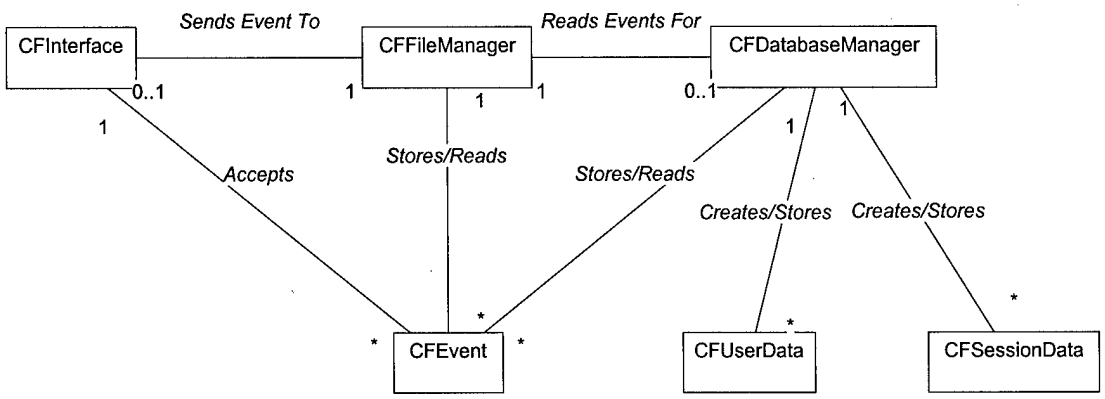


Figure 4-9. Framework Classes

4.4.4 The Core Framework Components

Information from the IDE is recorded by generating events. The events, in turn, are passed to a set of core framework components that put them to a more permanent form of storage. There are two basic configurations that the core components can take. In the first, they receive events from the IDE and store them to a disk file. This disk file is later read by a component which stores the events into a database. In the second mode, the events are queried from the database by the analysis tool. Figure 4-9 shows the core components and their relationships to each other.

Looking at the first mode, an IDE wishing to store events it has created, there are two core components that are used: CFInterface and CFFileManager. The IDE constructs an instance of CFInterface and then passes all events to it. CFInterface creates an instance of CFFileManager and uses this to store the events to a file. This file can later be read by a CFDatabaseManager instance and stored into a database.

While CFInterface could send events directly to CFDatabaseManager for storage, this can cause problems. First, if the system is hosted on an unstable network, the connection between the IDE and the database could be lost, resulting in a loss of data or the complication of the software by requiring extensive error-recovery procedures. Further, on a network of personal computers, it is possible the user may power off the computer rather than exit the program normally, resulting in a non-terminated session. The next problem can occur when considering the likely possibility that users will want to work at home, away from the network. This disallows a direct connection to the database and events must be stored in files.

In the second mode of operation, the core components are used by an analysis tool to extract data from the database. In this mode, the analysis tool constructs an instance of CFDatabaseManager and uses it to access the events. The database and its structure are hidden from the analysis tool through a set of application program interfaces (API) that the core framework components supply. This API is detailed in Appendix I. CFInterface is not used in this mode, and CFFileManager is only used if the analysis tool is also controlling the merging of event files into the database. The demonstration tool created for this dissertation works in this manner.

4.4.4.1 Application Interface (CFInterface)

The IDE uses the framework through a single interface class, CFInterface. The IDE creates an instance of this class and has it open an event file for writing. An event file may either be created, or it may be an existing event file in which case events are

appended to it. The IDE then creates instances of events and passes these events to the CFInterface object which performs any processing needed to get them written to disk.

CFInterface itself contains operations to open, create, and close event files. Other operations process the various events. The operations that process the CFEvtFileOpen and CFEvtFileNew events return a window handle that is used by subsequent window events to indicate what window in the environment they are operating on.

4.4.4.2 File Manager (CFFileManager)

The CFFileManager class has the responsibility of handling all operations dealing with event files. It is not used directly either by an IDE or by an analysis tool. It is a utility class used by CFInterface and CFDatabaseManager. It contains operations to create new event files, open existing event files to append events to them, and open event files to read events from them.

It is anticipated that the file manager could exist in a networked environment and act as a process on a server. In this role, it would need to be able to handle multiple, simultaneous file requests. To support this, the object returns a file handle from all of its file open and creation functions. This file handle is used by CFInterface to tell the file manager what file to write to when an event is appended. It also performs the same identification role when CFDatabaseManager requests an event.

CFFileManager has a single operation appendEvent for writing events to a file. It has a single operation getNextEvent which reads an event from an opened file. The getNextEvent operation returns a complete, instantiated event to the database manager.

The idea is to insulate CFInterface and CFDatabaseManager from the mechanics of how the file is read or written.

4.4.4.3 Database Manager (CFDatabaseManager)

The database manager has several roles. First, it must implement a method of storing events so that they can be pulled from the database in a variety of organizations. There are two operations for querying events: get events by user, and get events by session. Leaving the user identification empty retrieves a list of all events. However, since there is a strong possibility other organizations of the data may be needed, the database cannot be coded solely to meet these two operations. For this reason, it was decided that the CFDatabaseManager object would encapsulate a more general-purpose database and perform data conversions on the events to store and retrieve them from it.

The next major role of the database manager is to read event files and import the events into the database. To perform this, the CFDatabaseManager object instantiates an instance of CFFileManager and has it read an event file specified by the user. Events are extracted from the event file, examined for special processing by the database manager, and then inserted into the database. Some events require special processing because they contain information that needs to be stored separately. For example, as new users are discovered via CFEvtSessionStart events (which contain a user ID), there is a separate database structure that needs to be updated to add this user.

The handling of file-oriented events is particularly tricky. As indicated earlier, information is being recorded on an individual basis, not in a networked mode. Because

of this, when a user opens a new file and CFInterface returns a file handle to the user, that file handle is unique only to that particular session. It is not globally unique among sessions or among the community of users who may be putting events into the database. In order to uniquely identify the file being operated upon, the database manager replaces all local file handles with unique global handles when it reads in session events.

The final role of the database manager is to offer access to the underlying database system to the analysis tool. This access is to allow the tool to create database structures it may need to perform its job. The intent is that these structures would be unique and separate from the structures used by CFDatabaseManager. For example, in the analysis tool developed for this dissertation a separate table was needed to hold information about the state of charts and graphs being shown.

4.4.5 Database Design

One of the goals of the framework was to support tools that would allow managers to access a historical database of project information. In selecting the model of database to be used, one of the key decisions made was to use the relational model. This was done because of the large number of inexpensive relational database management systems (RDBMS) available, and the excellent support among object-oriented languages for Structured Query Language (SQL).

This choice has a drawback compared to using an object-oriented database however. Each event has to be converted by CFDatabaseManager into a tuple that is stored in a database relation. Extracting events means converting a tuple back into the internal event

object format. This required adding operations to events that caused them to convert their data back and forth as needed.

Figure 4-10 shows the structure of the database. The following sections describe the relations in the database.

4.4.5.1 Events Relation

This relation is composed of tuples that represent events. One tuple is one event. The fields within the tuple have the following meaning:

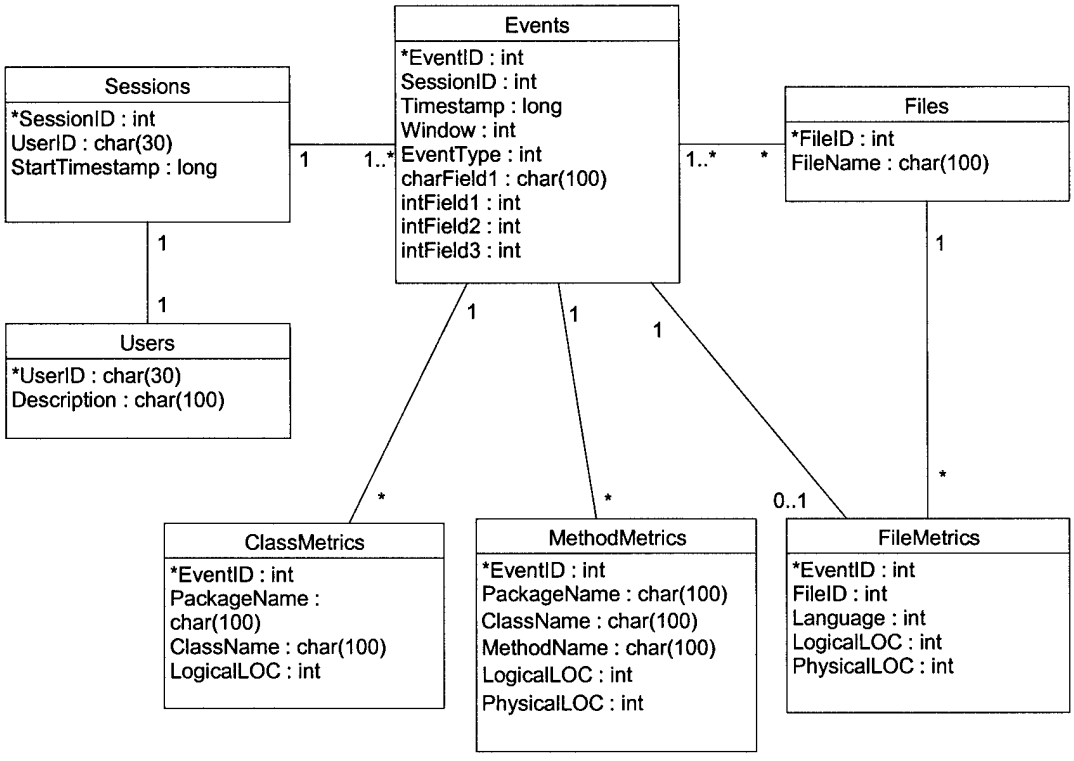


Figure 4-10. Database Relations

- *EventID (primary key).* This is an integer value assigned to the event by the database manager.
- *SessionID.* This field identifies which session the event belongs to and is used to access information in the Sessions relation.
- *Timestamp.* A long value which represents the time at which the event occurred. This value is stored as milliseconds since Jan 1st, 1970 (a typical epoch for date values).
- *Window.* A unique integer value that identifies what window or file the event occurred in.
- *EventType.* A value that tells the database manager what type of event is stored in the tuple. It uses this value when reconstructing events so that it knows what kind of object to instantiate.
- *charField1, intField1, intField2, intField3.* These fields store information unique to the kind of event specified by the field EventType. For example, an unsuccessful CSD generation stores the error text in charField1 while a file open event stores the filename in charField1.

The Events relation follows the representation pattern for tuples of S_d discussed in Section 4.4.3. The pattern there was:

$(t, \text{action\_name}, a_1, a_2, ..., a_n)$

With the Events relation there is:

(Timestamp, EventType, EventID, SessionID, ..., intField3)

As new attributes are required, the Events relation can be expanded with more attributes while still continuing to keep the basis of the framework's event design.

4.4.5.2 Sessions Relation

Each event must belong to exactly one session. A session may be composed of many events. The fields within each tuple of the Sessions relation have the following meaning:

- *SessionID (primary key).* An integer value assigned by the database manager that uniquely identifies this session.
- *UserID.* Each session is associated with one user. This field uniquely identifies the user and is used to access their information in the Users relation.
- *StartTimestamp, EndTimestamp.* These fields record when the session started and ended. While this information can be discovered by analyzing the events associated with this session, it is such a common task that it is precomputed and stored here to save time.

4.4.5.3 Users Relation

Every session is associated with a user. A user may have many sessions. The fields within the Users relation have the following meaning:

- *UserID (primary key).* A character field that uniquely identifies the user. This must be supplied in the CFEvtSessionStart event, or as a parameter passed to the CFDatabaseManager when an event file is imported into the database.
- *Description.* A character field that holds a description of who this user is.

4.4.5.4 Files Relation

As noted previously, when users operate on files the CFInterface object assigns local file handles for identifying them. When they are imported into the database, however, it is critical that there are no conflicts with duplicate file handles. Therefore, the database manager uses the Files relation to track what files (or windows) have been used, and assigns each a unique handle. The association between a global file handle/local file name is made in this relation.

The fields have the following meaning:

- *FileID (primary key).* An integer value assigned by the database manager that uniquely identifies this file or window. If the event contained a local file handle stored in the *events.window* field, it is replaced with this value.
- *FileName.* The name of the file as stored on the local computer system.

4.4.5.5 Metrics Relations

The relations FileMetrics, ClassMetrics, and MethodMetrics are used in conjunction with the CFEvtMetricsReport event. They store metrics computed for the file as a whole, the class, and each method in the class. Because they are intended to work for a variety of languages, it was decided not to assume that files contained classes and classes contained methods. Instead, it is possible to have a metrics report on a file which will insert a tuple into FileMetrics, not have any classes (in the case of the C language), but have several method metrics inserted into MethodMetrics. Because of these potential language differences, not all fields may be used in each relation's tuple.

Metric tuples are keyed upon the CFEvtMetricsReport event that created them. So a single event may have no metrics associated with it, or if it is a CFEvtMetricsReport event it will have many metrics tuples associated with it.

4.5 IMPLEMENTATION

Section 4.4 discussed the architecture of the system and described the components and their relationships, but avoided discussion of specific implementation of the relationships or operations. In this section, the framework architecture implementation is discussed.

4.5.1 Language Selection

Selecting a programming language to implement the framework in was a key task. The language candidates had to be considered under the following criteria:

- *Object-Oriented Support.* Object-oriented programming can be performed in most procedural languages. If the language is not inherently object-oriented, then the key concepts of classes, inheritance, and polymorphism have to be simulated through data structures, hierarchy flattening, and large case structures. If the language has built-in constructs to support classes, inheritance, and polymorphism, then the amount of code required is reduced.
- *Portability.* It was important that the language be available on multiple platforms, and that code written in that language work on different platforms without code change. This requirement stems from the expectation that experiments and industrial use will occur both in Microsoft Windows and Unix environments.

- *Database Support.* The framework needs a relational database for persistent storage of events. Languages with capabilities such as embedded SQL or powerful SQL application program interface (API) libraries will support this easier than those which require custom coding to support a database.
- *Popularity.* Obscure languages are not going to be useful since it is unlikely that they will interface with environments being developed for commercial use.
- *Ability to Interface with GRASP.* One of the key goals of this research is to support GRASP. GRASP is currently be rewritten in Java, so the chosen language would either need to be Java or be supported by it via a native interface.

There were three main candidate languages considered:

- Ada95. Ada95 was a strong candidate since it directly support object-oriented concepts, is very portable, and could interface with GRASP through bindings. Support for inexpensive databases, however was lacking. Ada95 also lacks the new development popularity of the other languages considered.
- C++. This language was a strong candidate in all the categories. It offers direct support of object-oriented programming, has excellent database options, very popular, and could be integrated with GRASP. Portability is somewhat problematic since vendors tend to customize their compilers slightly. However, sticking to the draft ANSI standard could overcome this. A major drawback of using C++ is that unlike Ada95 or Java, there is not a lot of strong typing or built-in program safety options. This makes faulty code more possible in this language.

- Java. Like C++, Java is also a strong candidate in all the categories. Java is an object-oriented language very similar to C++, has many database support options, and easily integrates with the new version of GRASP being written in Java. Additionally, it supports a stronger data type model than C++ and has automatic garbage collection. The major drawback of Java is its speed. Java is compiled to platform-independent bytecode and this bytecode is executed via a virtual machine. Optimizations by vendors continue to improve Java's speed, however.

Because of Java's support of the features, and the ability to natively interface with GRASP and other new IDEs being developed, it was selected as the language for the framework. Sun Microsystems currently maintains the Java language specification, and issues the reference Java Development Kits (JDKs). The framework was developed using Sun Microsystem's JDK 1.2.2.

4.5.2 The Event Hierarchy

Java offers the capability of specifying "packages" of classes. A package is a grouping of classes that either have similar functions, or they have a logical reason to be grouped together. For example, there are Math packages for mathematical functions, the Abstract Window Toolkit (AWT) for constructing simple graphical user interfaces, etc. The event hierarchy was placed into a Java package called *cframework.events*. An application gets access to these events by adding the line "`import cframework.events.*`" in the start of the code.

Each event class was placed in a separate file. Because Java always performs dynamic class loading, this allows a class to be changed, separately compiled, and placed back into the system without requiring any other classes to be recompiled. Other classes only need to be recompiled if the interface signature on a class's methods change.

The events themselves are essentially data structures, so the only methods implemented in them were accessors and mutators for the class variables. Each event also has the method toString() implemented so that it can easily be printed for debugging output. The toString() method prints the internal state of the object to a String variable.

4.5.3 Core Framework Components

The core application components were placed in the package *cframework*. As with events, each component is stored in its own file which allows separate compilation.

4.5.3.1 Application Interface

The class *cframework.CFInterface* is the implementation of the application interface. The mechanics of the class are implemented as specified in the previous architecture discussion. The IDE creates a single instance of this class. The implementation of the class provides an overloaded method called event(). This method is overloaded for every allowable event type. This is necessary so that individual processing can be done on events as necessary. Providing a single, non-overloaded method that accepted anything of type CFEvent would have meant creating a large case statement in the method. This

would have made the code unnecessarily complex and would have avoided a big advantage of object-oriented programming – polymorphism.

The methods that handle CFEvtFileOpen and CFEvtFileNew keep track of local file handles. When one of these events is passed to the event() function, the object examines its list of open files for an empty slot. It then returns the index of this slot as a file handle. In subsequent operations, this file handle is placed into events by the IDE when it creates them. An example of creating an application interface, opening an events file, and inserting events is shown in Figure 4-11.

4.5.3.2 File Manager

The class *cframework.CFFileManager* is a utility class used by the file manager and the database manager, it should never be created or accessed directly by an application or analysis tool. As specified in the architecture, it contains the methods needed to create a new event file, open an existing event file for the purpose of appending to it, and open an existing event file for the purpose of reading from it.

CFFileManager is written so that it can eventually be placed into a network server process. It does this by returning a file handle when an event file is opened, allowing more than one file to be open at any time. When writing events, the file handle is passed as a parameter to tell the file manager what event file to write events to.

```
import java.io.*;
import cframework.*;
import cframework.events.*;

class InterfaceTest
{
    public static void main(String [] args) throws IOException
    {
        CFInterface evtInterface;
        CFEvent event;
        int handle;

        // Create an interface
        evtInterface = new CFInterface();
        // Create an event file
        evtInterface.createFile("events.log");
        // Start a session
        event = new CFEvtSessionStart("User01");
        evtInterface.event((CFEvtSessionStart)event);
        // Simulate opening a file in the editor
        event = new CFEvtFileOpen("TestFile.java");
        handle = evtInterface.event((CFEvtFileOpen)event);
        // Simulate closing the file
        event = new CFEvtFileClose(handle);
        evtInterface.event((CFEvtFileClose)event);
        // End session
        event = new CFEvtSessionEnd();
        evtInterface.event((CFEvtSessionEnd)event);
        // Close event file
        evtInterface.closeFile();
    }
}
```

Figure 4-11. Example of Using CFInterface

Events are written to files using the method appendEvent(), and read from files using the method getNextEvent(). If there are no events left to read from the file, getNextEvent() returns the value NULL.

Java has a very robust file system allowing many different storage formats. CFFileManager has been set up to allow optimized file handling for Java IDEs, and portable file handling for non-Java IDEs. This is done by having two distinct storage formats: binary and ASCII.

BINARY EVENT FILES

Java's file system allows objects to be written directly to files using a technique called object serialization [Campione and Walrath 1998]. This technique lets a program open a stream of type ObjectOutputStream and write objects directly to it. The objects being written must have implemented the interface Serializable. The interface Serializable, however, contains no methods. It merely allows the object writer method of the output stream to work with the class.

The ObjectOutputStream will automatically write all the fields of the objects to the stream, and handle transient and non-transient values. An example of a transient value would be a reference to another object. The reference must be written in such a manner that when the object is read back from the stream, the reference correctly points back to the other object. This implies the other object exists, which implies that the serialization scheme must know the correct order to create objects when reading them back out. Java's implementation of this mechanism handles this automatically making it a simple task to write events to a file.

Because a large number of events can be written to the file, a compressed character stream is attached to the ObjectOutputStream. This is done by sending the output of ObjectOutputStream to an instance of DeflaterOutputStream. This compresses the data before passing it to an instance of FileOutputStream which writes the compressed data as characters to a file.

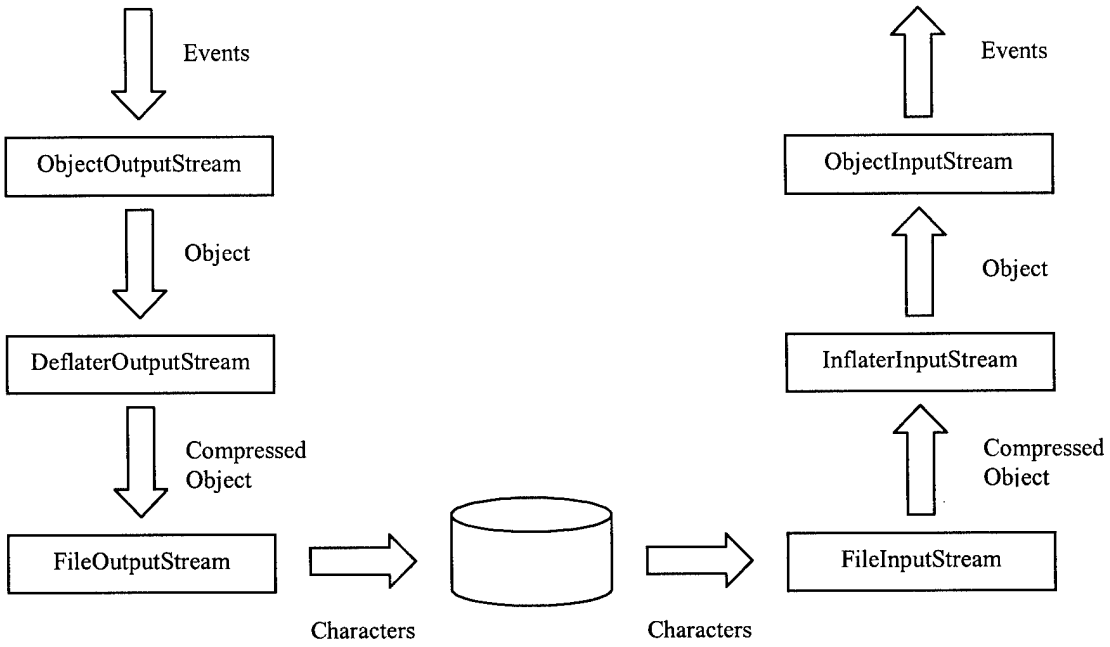


Figure 4-12. Event Writing/Reading Processes

When the file manager reads events from the file, there is a reverse of this process. An instance of FileInputStream reads compressed information, which is passed to an instance of InflaterInputStream. The InflaterInputStream object uncompresses the data and hands it to an instance of ObjectInputStream which converts the data back into an object. The entire process of writing and reading objects is illustrated in Figure 4-12.

ASCII EVENT FILES

One of the problems with using binary files is that only an IDE written in Java can use them. Development environments such as pcGRASP are written in C++, and cannot directly instantiate and use a CFInterface object to write event files. Because of this, the file manager accepts an alternate form of event file that is composed of ASCII text.

In this case, the development environment itself is responsible for the creation of the event file. The basic format for each event in the ASCII file is the time of the event in milliseconds since a defined epoch (usually Jan 1st, 1970), followed by the type of event. Unique information about the event follows. The specific format of each event is specified in Appendix A.

The file manager uses a different set of methods to read ASCII files. An ASCII file is opened for reading using openFileASCII() instead of openFile(). Instead of using getNextEvent() to read an event, the method getNextEventASCII() is used.

4.5.3.3 Database Manager

The database is managed through the class *cframework.CFDatabaseManager*. The nature of this task makes the database manager the most complex of the framework components. It contains methods to create new databases, process event files, return event lists, return session lists, and return user lists.

The database manager is used by an analysis tool. This is done by instantiating a single instance of CFDatabaseManager. The constructor for the object creates a connection to the database and prepares it for use. To accomplish this, the user must specify the location of the database via a unified resource locator (URL), a username, and a password. CFDatabaseManager creates the connection to the database, and then logs in to it using the supplied username and password. If there is a problem, it throws a Java exception called *SQLException*.

JAVA DATABASE CONNECTIVITY (JDBC)

A major goal of the framework was to remain platform-independent. This means that not only should the language the framework is written in be portable, but the database management system it uses should also be portable. This gives the user of the framework the flexibility to move the applications across operating systems without worrying about the underlying database no longer working.

Java has a big advantage in this area not only because the language itself is portable, but because it uses a portable toolkit for manipulating databases called the Java Database Connectivity (JDBC) kit [Cornell and Horstmann 1997]. Rather than custom-coding a database for the framework, we use the API available through JDBC to access the underlying database. Because of the way in which JDBC works, the database does not have to be written in Java. It merely has to communicate with Java through a JDBC driver, or through a special driver known as a JDBC/ODBC bridge (ODBC is Microsoft's Open Database Connectivity kit). Figure 4-13 illustrates the connections.

A JDBC driver supplies four major pieces of functionality:

- *Database Connectivity.* JDBC allows users to open a connection to a relational database. This database can be hosted locally, or it can be made available anywhere on the network. It is accessed by passing a URL to the driver. This means that while the framework might be running on a Windows 98 computer, the database may be running elsewhere on a Unix machine. JDBC makes this transparent to the user of the API.

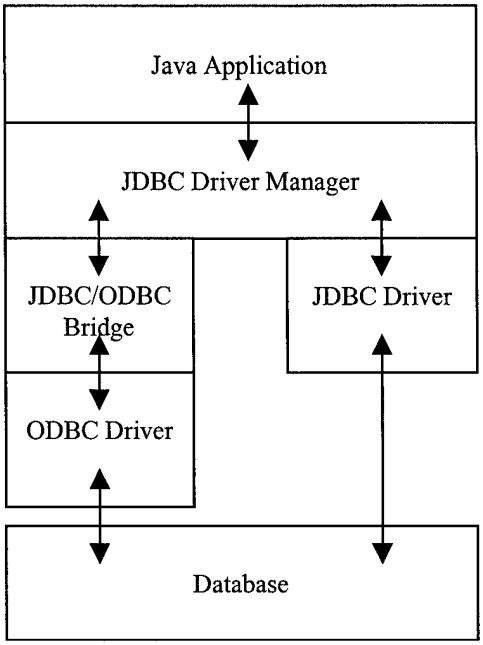


Figure 4-13. JDBC Connectivity

- *Data Definition Language (DDL) Commands*. DDL allows us to set up the database's data representation scheme. In a subset of SQL are the DDL commands, the ones most commonly used are the CREATE TABLE, DROP, and CREATE INDEX commands. JDBC supports these through a simple method called executeUpdate() which accepts the command as a string. This method accepts any command which does not return a result set. For a detailed discussion of DDL, see [Korth and Silberschatz 1991].
- *Data Manipulation Language (DML) Commands*. DML allows us to manipulate the data in the database. The most commonly used SQL commands that do this are the UPDATE and SELECT statements. Updates to the database are accomplished using the executeUpdate() method since updates do not return a result set. Queries on the database via the SELECT command, however, use the

method executeQuery() which passes the SELECT statement via a string. The method returns a special structure which contains the results of the query. For a detailed discussion of DML, see [Korth and Silberschatz 1991].

- *Metadata.* A useful capability of JDBC is its ability to describe the scheme of the database. That is, JDBC can return the names of the fields in a table and describe their representation. While this capability is not used by the framework, it is useful for applications which may not have advanced knowledge of the database's organization.

INSTANTDB

There are a wide variety of databases available that run under JDBC. To select one, several factors were considered. First the size of the database system should not be excessive in terms of memory and disk space used. Second, the database system should be easy to use and setup, and not require extensive training to maintain. Third, a preference was to have a database that was distributable with the framework, or that could be obtained for minimal cost from a web site. Finally, while there was not an extreme access speed requirement, the database system performance should not degrade as more data was added.

The database selected for use with the framework was InstantDB, developed by Instant Computer Solutions, Ltd., London, England, U.K. InstantDB has a very small footprint, requiring only 160k of disk drive space for the class libraries, and only a simple text file on the hard disk to describe the database's characteristics. The cost is free to

non-commercial and non-government users, and is very reasonable to all others. InstantDB implements nearly all of the JDBC standard, and supports most of SQL '92.

A major advantage of InstantDB is that the fact that it is written entirely in Java. This means that not only is the database itself portable, but the files that it creates are also portable since Java stores data in a platform-neutral format. Thus a database created on a Windows 98 machine can be sent to a Unix machine and still be read and manipulated by the framework without translation between word formats. Some machines put the most significant bit at the beginning of a word, others place it at the end.

The major disadvantage of InstantDB is its speed. While Java's speed is rapidly approaching that of equivalent C or C++ code, it is still slower. Databases written in non-interpreted languages will likely have better performance. The framework has been implemented not to depend on InstantDB and allows the easy integration of alternatives such as MySQL, mSQL, Oracle, Sybase, or Interbase. This can be done by changing the connection string in the CFDatabaseManager's constructor.

DATABASE CREATION

CFDatabaseManager's constructor establishes a connection through JDBC to InstantDB. InstantDB is given the location of a database properties file that describes where the database is located and the settings of various performance tuning parameters on it. If InstantDB determines the database files do not exist, it creates them and opens the database connection. Note that this is a unique feature of InstantDB, other database systems such as MySQL or mSQL require that the database files exist. If one of these

databases is eventually used in place of InstantDB, the CFDatabaseManager constructor will throw a SQLException if the open fails.

The act of creating the database files does not establish the relational scheme within the database itself. CFDatabaseManager supplies the method createDatabase() which drops any existing framework tables and creates new, empty tables. These tables include those defined in Section 4.3.4, plus three additional tables that are used to support generation of unique identifiers: EventIDSequence, FileIDSequence, and SessionIDSequence.

EVENT INSERTION

Event files are loaded into the database using the processEventFile() method. This method takes three parameters: the name of the file, a boolean flag set to true if this is a binary file or false if it is an ASCII file, and an optional username to associate with each session in the file. Depending on the value of the file type flag, processEventFile() will tell CFFileManager to open a binary event file or an ASCII event file. It also examines this flag when determining to call CFFileManager's getNextEvent() or getNextEventASCII(). Beyond that, processing is the same for both file types.

As events are read from the event file, the database manager watches for session start events. If it finds one, it assigns it a unique identifier and records it in the sessions table. Each event read out of the file after that is marked with that session identifier until either a session end event is found, or another session start is encountered. If another session

start is encountered, this usually means the user did not properly close the environment down.

Each event is inserted into the database in the method insertEvent(). This method queries the event to extract the session, time, window, charField1, intField2, intField2, and int Field3 fields for the database tuple that it creates. It then inserts this tuple into the Events table.

Some events require special processing. Metrics events have embedded information that must be stored in separate tables as described in Section 4.4.4.5. Events that involve opening, creating, or saving files require manipulation of the file handles. Events that operate on files always have a local file handle assigned by CFInterface. The insertEvent() method gets a global handle from FileIDSequence and substitutes it for the local handle. This ensures there are no handle conflicts in the database.

DATABASE QUERIES

CFDatabaseManager returns events via two basic queries: by event type or by session. The method getEventList() is overloaded so that one version accepts an event type and username. The other version accepts a session identifier. In the first version, it queries the database for all events of the specified type that were created by the specified user. If the username is NULL, then all events of the specified type are selected. If the event type is of type EVT_ALL (defined in the class CFEventMap), then all events are selected.

In either case, getEventList() constructs an SQL query in a String. This string is then passed to the JDBC executeQuery() method which returns a ResultSet object. Each tuple in the result set is converted back into an event using the method createEvent(). The event is then add to a Java collection called a Vector, which operates like a linked-list ADT. After all tuples in the result set have been processed, the Vector is returned to the caller. The methods getSessionList() and getUserList() return a Java Vector containing all sessions and all users in the database, respectively.

EXTERNAL DATABASE ACCESS

It is expected that any analysis tool that is written for the framework may require the use of a database to support its operations. CFDatabaseManager contains a method getCon() which returns a JDBC connection to the database. Analysis applications can use this connection to create tables and populate them with data. The intent, however, is not to give access to the tables used by the framework. Analysis applications should not directly manipulate framework data via this connection.

CHAPTER 5. CASE STUDY: INTEGRATION WITH PCGRASP

5.1 INTRODUCTION

This chapter discusses how the framework was integrated with pcGRASP in order to support comprehension experiments using CSDs. Background on the history of GRASP is given, followed by a discussion of how pcGRASP fits into the project. This is followed by a discussion of the necessary enhancements to pcGRASP to use the framework, and the design and construction an anlysis tool to read the results. Finally, the results of a series of end-to-end tests are presented.

5.2 PCGRASP

As noted in Chapter 2, GRASP is an NSF-funded project at Auburn University that seeks to develop practical visualizations of algorithms, structures, and processes for use in solving real-world software engineering problems [Hendrix *et. al.* 1998]. The primary emphasis to date has been on the CSD. There are two main software development environments that have been developed to support this: Unix GRASP which runs under Unix/Linux environments, and pcGRASP which runs under Microsoft Windows NT/95/98. Figure 5-1 is a screen capture of the pcGRASP environment.

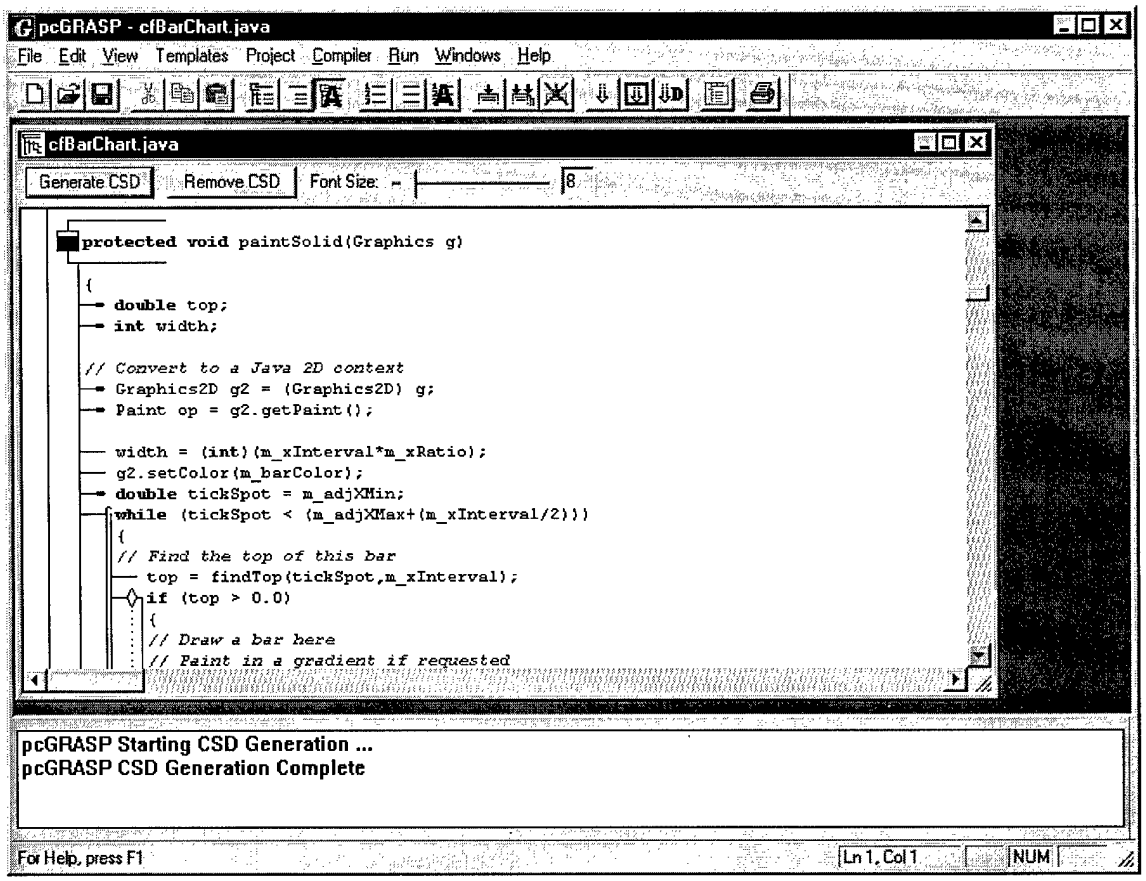


Figure 5-1. pcGRASP Main Screen

5.2.1 Background

The prototype tool developed to support GRASP was originally written in Ada and ran under Unix. While it was innovative in the diagrams it could present to the user, it suffered from poor CSD generation times. In 1995 work was begun to rewrite GRASP in C to enhance peformance and provide greater compatibility with the X Windows graphic user interface architecture. The result was a tool that was very user-friendly and suitable for industry use, with CSD generation rates of over 5000 lines per second [Cross *et. al.* 1996].

GRASP, which originally supported CSD diagrams only for Ada, was subsequently expanded to include C, C++, VHDL, and in 1998, Java. As part of the effort to improve the visualization environment, GRASP was enhanced to allow syntax highlighting, color, and font selection through the use of a CSD editor. GRASP is currently being rewritten in Java to maximize its availability to users on different platforms.

5.2.2 Purpose

On of the drawbacks of the GRASP was that it was confined to the Unix environment, which for most programmers meant it was available only at school or at work. In work environments that used only Microsoft Windows operating systems, it was not available at all.

pcGRASP was created in early 1997 to address the need for a version of GRASP that could run natively in the Windows environment. This allowed users to generate CSDs outside the work and school environment. It was also an easy way to deliver GRASP to colleges and universities that maintained labs consisting mostly of personal computers.

5.2.3 Design and Implementation

The basic design philosophy behind pcGRASP was to create a Windows interface that encapsulated the Unix CSD generator. Eventually, the CSD editor was ported over from the Unix version of GRASP and integrated into pcGRASP. Thus, the major subsystems of the environment are: the graphic user interface (GUI), the CSD editor, and the CSD generator. The following sections discuss the pcGRASP architecture in greater detail.

5.2.3.1 Microsoft MFC Document-View Architecture

The lowest level API for communicating with windows is via the Win32 interface supplied by Microsoft. This interface consists of several thousand functions and can be quite daunting to even experienced programmers. Even when mastered, developing applications from scratch can take a considerable amount of time.

To ease the development of Windows applications, Microsoft created an application framework based on the Model/View/Controller classes from Smalltalk [Kruglinski 1996]. Known as the Microsoft Foundation Classes (MFC), they are a large group of components that can be assembled to form a generic application. The programmer controls how the components are put together, and adds the unique code required to create a useful program.

In a typical MFC application, there are usually three main objects that do most of the work. The application is responsible for starting the program and shutting it down, as well as opening documents. Documents are non-visible constructs that hold information that is being manipulated. Views are the visible element that a user works with to change the data in a document. There can be multiple views on a single document, and multiple documents open in a single application. Ancillary objects control other items in the user interface and are usually embedded into an application, document, or view object.

5.2.3.2 pcGRASP Core Components

pcGRASP follows the MFC document-view architecture, and adds three major components. The application object is CGraspApp, the document object is CGraspDoc,

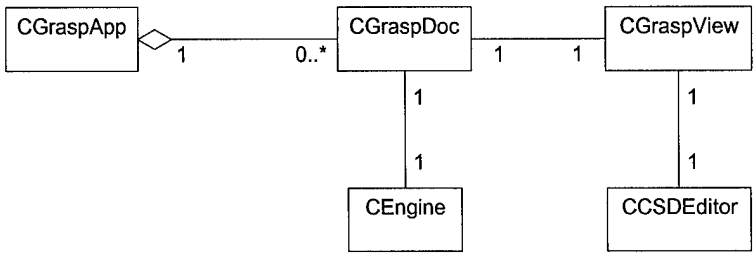


Figure 5-2. pcGRASP Components

and the view object is CGraspView. The additional components are: CEngine, which controls the generation of the CSD and compiling; CCSDEditor which is an editor designed for use with CSDs, and a set of C routines which generate the CSD. The C routines are the Unix CSD generation routines compiled under Microsoft Visual C++. Figure 5-2 shows the relationship between these components.

5.2.3.3 Document Engine and CSD Generator

The CGraspDoc object in pcGRASP holds the characters that represent the source file being edited. When a user requests that a compile take place, or that a CSD be generated, the document has to invoke the correct compiler, or call the correct CSD generation routine depending on the language the source code is written in.

To accomplish this, each document object is associated with a child class instance of CEngine. CEngine knows how to correctly invoke the CSD generator and how to invoke the compiler. This knowledge of what to call is created by subclassing CEngine once for each of the major languages supported. Thus, the Java version of CEngine is called CEngineJava and the methods for compiling and CSD generation are overridden.

As noted in the previous section, the CSD generator itself is the Unix C code compiled under Microsoft Visual C++. About half the code is the output of a grammar run through Bison (a parser generator utility). A separate grammar is available for each language, and the resulting code from each grammar is invoked by a C function specific to that language.

5.2.3.4 CSD Editor

Each CGraspView object contains a single instance of CCSDEdit. CCSDEdit is a Windows conversion of the Unix CSD editor. This editor is unique in that it is sensitive to the existence of a CSD diagram in the editor window, and adjusts the diagram as needed when users make changes. It ensures the cursor does not end up on CSD characters, which means the programmer is free to work on the code without worrying about deleting the CSD or somehow distorting it. In addition, the CSD editor is context sensitive to the language being edited and will highlight key words, operators, strings, etc. as desired by the user.

5.3 ENHANCEMENTS TO PCGRASP

In order to use the framework with pcGRASP, there were several major changes that had to be made to the software. First, the code had to be instrumented so as to generate the appropriate event when a user peformed key actions. Second, the parser that generates the CSD had to be modified to collect software metrics during the parsing process. Last, the software had to be augmented to include code that could properly output event files.

5.3.1 Event Instrumentation

Event instrumentation in pcGRASP is spread across the code in many places due to the nature of the data being collected. In general, however, line-oriented events are caught by the class CCSDEdit. CSD and compile events are recorded by CGraspView. CGraspApp catches session start and end events, along with all opening and closing of windows.

A dialog was added to the preferences sheet of pcGRASP that allows control of the collection process. Specifically, users may specify the event file to write and activate or deactivate event collection. In addition, since some events will probably not be needed, the user may select which type of events should be collected: CSD generations, compilations, window activations, file executions, and software metrics. Figure 5-3 shows the dialog.

A menu item was also added that allows users under PSP to specify what phase of a project they are working on. It also includes menu items to indicate their work has been interrupted by phone calls, conversation, or personal breaks.

5.3.2 Java Parser Enhancement

As noted previously, pcGRASP has high-performance parsers that assist in generating the CSD. These parsers are created by passing a grammar file through the GNU Bison parser generator. Each language has its own grammar file and resulting parser.

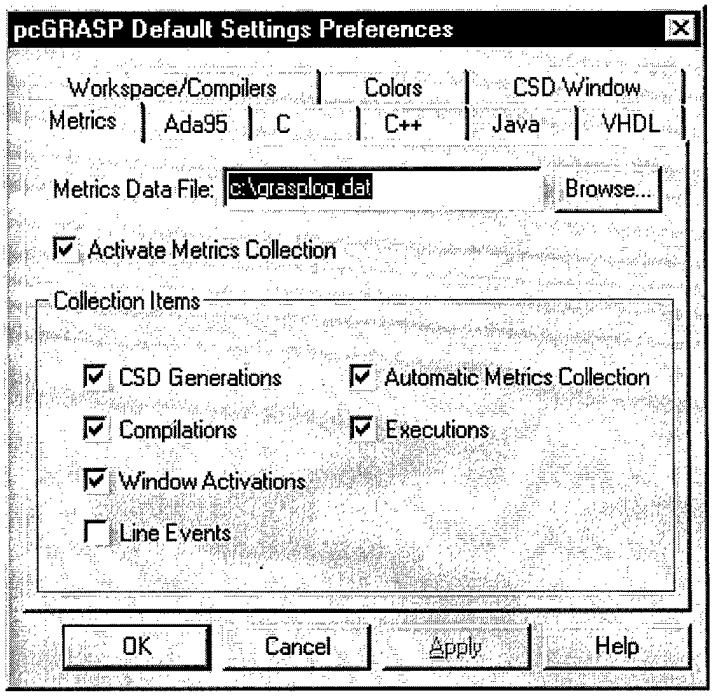


Figure 5-3. pcGRASP Collection Dialog

In order to demonstrate the collection of software metrics, and to assist with classes at Auburn University which are going to begin utilizing Java, the Java parser was modified to collect logical and physical lines of code. It was also modified to collect cyclomatic complexity.

Physical lines of code were defined as the number of physical lines in the file. Logical lines of code were counted using the SPR method defined in Section 2.3.2.1. Some modifications were made to this so that class declarations are counted as a line, and method declarations are counted as a line. A package declaration, if present, is also counted as one line.

Cyclomatic complexity is computed by counting the number of predicates (conditions) in the file and adding one. This corresponds to the method specified by McCabe in Section 2.3.3.1. This is computed on a method-by-method basis within each class.

In order to minimize the overhead of collecting these metrics, no computation takes place inside the parser. Instead, an array of characters is passed to the parser – one element for every character in the file. As constructs are discovered by the parser, this array is modified to indicate where packages, classes, and methods start/end, where conditions start/end, and where logical lines of code end. After parsing, the array is processed by a routine which computes the actual metrics and puts them in the form required by the software metrics event.

5.3.3 Output Format

The main problem with integrating pcGRASP into the framework stemmed from its implementation language, C++. This meant that it could not directly write a binary event file that Java could read properly. This left two possible options: invoke a separate process that would start a Java virtual machine and communicate with it via sockets, or develop an ASCII file format for writing events and have the framework read that.

The socket method would have been problematic for users of Windows who were not on a network, since it would have required them to install network software they were not actually going to use. In addition, this would require the user to download the correct version of the Java Runtime Environment (JRE) and install it on their machine.

Because of these issues, pcGRASP outputs the events in ASCII to a file. The format of this file is in Appendix A, and the mechanics of how the framework reads it is described in Chapter 4.

5.4 ANALYSIS SYSTEM

Key to demonstrating the usefulness of the data collection framework was an analysis tool that could show how correlating time with measurements produces meaningful results. A tool called the "Collection Framework Analyzer" (CFA) was developed for this purpose. It is capable of controlling the import of event files into the database manager, organizing the data in terms of research and project groups, and displaying the data in report and chart form. While it has been specifically tailored to work with pcGRASP data, its architectural concepts leave it open to enhancement for other kinds of environments.

5.4.1 CFA System Requirements

As noted in the introduction in Chapter 1, the main problem researchers and industry share is collecting data on groups of individuals. A key concept in that problem is that the data is organized by group. The analysis of that data comes from looking at reports on a group's activities, or at a graphical representation of that activity in the form of a chart. The idea of data, reports, and charts being organized around a group is a core concept in the design of the CFA.

5.4.1.1 The Group Concept

CFA organizes information in terms of groups. This reflects the nature of research in which individuals are organized into groups for an experiment. In industry, it reflects the idea that projects are usually executed by a group of people. Support for Humphrey's PSP is allowed by assigning only one person to the group – the programmer tracking their statistics.

As Figure 5-4 shows, the group is the focal point for organizing and accessing information. Note that individuals, sessions, and events can be present in several groups. This way a researcher could define a group consisting of all test subjects, then create two additional groups that represent the novices and the experts from the overall subject pool. Industry users will usually have people who are working on more than one project, in which case they will have programmers in several project groups.

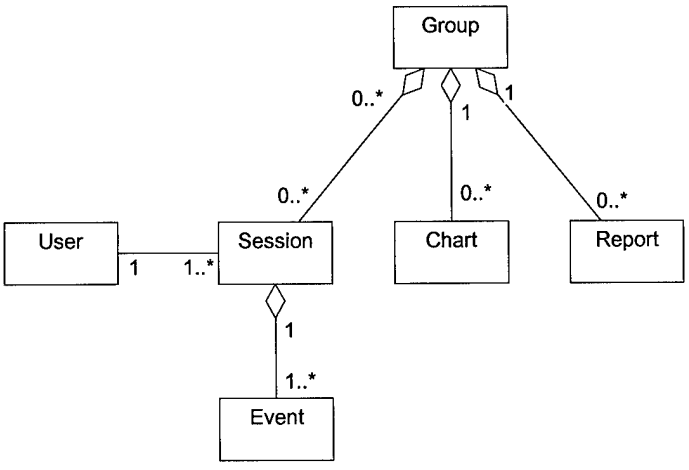


Figure 5-4. Group Associations

5.4.1.2 Report Requirements

In order to support a variety of anticipated needs, several types of reports are possible in CFA:

- *Session Summary.* A summary of all sessions for the group that lists the start date/time of the session, the amount of time in the session, number of files opened, number of compiles, number of compile errors, number of successful and unsuccessful CSD generations, and percentage of session time the CSD was used.
- *Session Detail.* A listing of all events in a specific session in the order they occurred.
- *User Summary.* A summary of each user's activity in a group.
- *User Detail.* A detailed listing of a user's sessions and total time spent in each development phase as specified in the CFEvtPhase event.
- *File Summary.* A listing of each file a group has used, summarizing the activity on the file.
- *Software Metrics.* A report for each file a group has used showing the latest information known about its size and complexity organized by package, class, and method.

Because these are only a small set of what will probably be a large suite of reports, CFA has the ability to easily add new reports as required.

5.4.1.3 Chart Requirements

In addition to the reports above, there are a variety of charts possible:

- *File Activity.* A pie chart showing the amount of time a file was open and active over all sessions in the group.
- *File Activity Over Time.* A bar chart showing the amount of time a file was open and active over a given interval of time. This is useful for determining if most of a file's use was at the beginning, end, or middle of a project.
- *Events Over Time.* A bar chart showing the events recorded in a session over a given interval of time. Useful for determining things like when the most compiles occurred or when users are activating the CSD.
- *CSD Use Over Time.* A surface plot of the fractional portion of a session that a CSD was active over a given time interval.
- *File Growth Over Time.* A plot of the amount of logical or physical lines in a file (or files), over time.
- *File Complexity Over Time.* A plot of the total cyclomatic complexity of a file (or files), over a given interval of time.

Like reports, CFA has the ability to easily add new charts as required.

5.4.2 Analysis System Design

The basic design of the CFA divided it into four main subsystems: the application, groups, group display items, and display item user interface components. Figure 5-5 shows the relationships between these components.

The design considers that users will need to define groups and then assign display items to them in the form of reports and charts. Each of these display items will have a unique configuration of data and preference settings for that group. These settings are stored in the database so they are persistent between sessions.

5.4.3 The Application Object

The application object, called Analyzer, is responsible for coordinating all major tasks. Upon startup, it creates an instance of CFDatabaseManager and opens the database. It reads all groups from the database and allows the user to manipulate them. It allows the user to create new groups. It allows the user to import both binary and ASCII event files into the database.

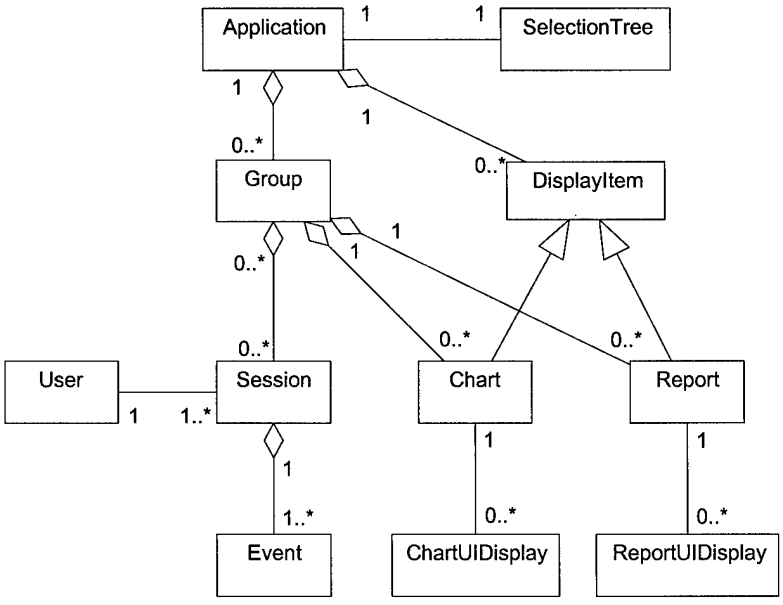


Figure 5-5. CFA Class Diagram

5.4.3.1 The Group Object

The group object, called Group, maintains a collection of sessions it is associated with, and a collection of group display items. All access to charts and reports is through a Group object, and it is responsible for ensuring they are correctly written and retrieved from the database. When a group is modified, it ensures that all displayed reports and charts are updated. When a group is removed, it ensures that all associated reports and charts are also removed.

5.4.3.2 Reports and Charts

As shown in Figure 5-5, the classes Report and Chart implement the interface GroupItem. The idea of an "interface" is somewhat unique to Java. A simplified explanation would be to consider it a type of inheritance: an interface can be considered a parent class where all methods are abstract and virtual, and therefore must be implemented.

The GroupItem interface allows a Group object to abstractly handle its charts and reports. Group display items can be told to return a displayable user interface element (the chart or report), close the user interface element, set user interface settings, and refresh the user interface element.

The handling of the display data and the handling of the user interface are kept separate. Subclasses of Report and Chart do not directly manipulate the display. Their purpose is to communicate with the CFDatabaseManager instance to collect and organize the information that needs to be displayed. To display the information, they create

instances of cfChart or cfReport which directly manipulate the Java GUI to display a variety of charts and reports.

5.4.4 Chart Package

As noted in the previous section, different objects handle data manipulation and user interface manipulation. This was done to ensure that if better user interfaces became available, the tool could be adapted to use them. Likewise, if better database access methods become available, the user interface routines could still interface to them.

A major problem, however, was the lack of general purpose chart and report packages for Java. A search of the Internet revealed several commercial packages available, but most were not written in the current version of the Java 2D graphics library and had poor appearance. The package that was written using the Java 2D graphics library was somewhat costly and did not contain many required features.

To solve this problem, a package of generic chart classes was written using the Java 2D graphics library. This library was distributed starting with JDK 1.2 and contains a robust set of primitives that allow users of it to specify line thickness, line end caps, line joints, line dash patterns, gradient patterns in polygon fills, a rich set of font operations, and affine transformations on all shapes and fonts.

The package, whose structure is shown in Figure 5-6, has the base class cfChart. There are two subclasses of cfChart representing major chart types: cfBarChart and cfLineChart. cfChart is capable of accepting multiple data series for display, where a data series is an array of x, y values encapsulated by a cfDataItem object. It is

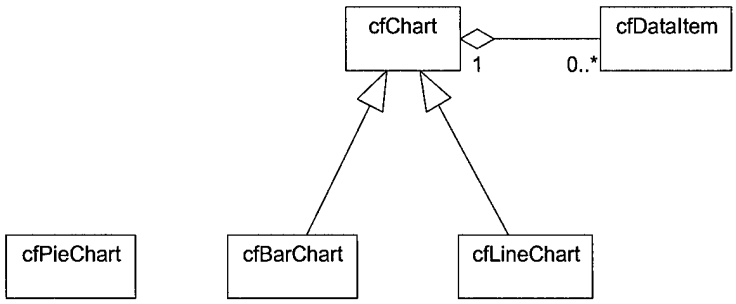


Figure 5-6. Chart Package Class Diagram

responsible for drawing the chart titles, axis, labels, grid lines, and background gradients. For the x axis, it can display numeric labels computed from the series itself, or display string labels supplied by the user. It can rotate the labels 90 degrees if requested by the user. As the size of the chart is changed by the user, it determines how many labels can be displayed and removes them as necessary to eliminate crowding.

cfBarChart displays a conventional vertical bar chart. It can display both solid bar charts where all the values of the data series are totaled into a single value for each bar, or it can display stacked bar charts where each data series value is shown separately as a segment of each bar. For solid bar charts, the bars can be colored in a gradient. Negative values are allowed, and the axis will be adjusted accordingly to display bars going down instead of up. Samples of a solid bar chart and stacked bar chart are shown in Figure 5-7.

cfLineChart is capable of displaying line plots, surface plots, or scatter plots. For line and scatter plots, symbols may be plotted at each data point. Currently available symbols are squares, circles, and triangles in three different sizes. For line and surface plots, the line or surface area color can be specified for each data series. Figure 5-8 contains examples of line, surface, and scatter charts.

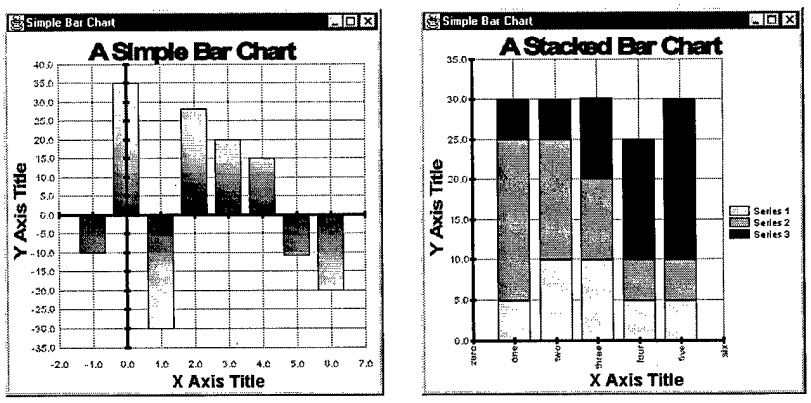


Figure 5-7. Solid and Stacked Bar Charts

cfPieChart does not inherit from cfChart. The reason for this is the inherent differences in how data is organized and displayed. cfPieChart keeps a single data series where each item in the series may be assigned a unique color and label. If a color is not specified, the chart assigns one. Each element in the data series has a flag that indicates whether it should be "cut-out" from the pie. If this is set to true, then the segment for that element is translated a small distance from the center so that it looks as if it had been cut and moved outward. The effect is to highlight that value to the viewer. An example of a pie chart with a cut-out value is shown in Figure 5-8.

The above object classes are not tailored to CFA or the data collection framework. Any application can instantiate and use them. Since they inherit from JPanel, they can easily be embedded into other user interface elements.

5.4.5 Implementation

As noted in the previous section, the main application object is the Analyzer which is a subclass of Java's JFrame. A JFrame is the basic window that exists in X Windows

under Unix or in Windows 95/98/NT. It provides a border, a title bar, a menu, and controls for sizing/closing the window. In Java's Swing library (one of two available GUI APIs under Java) applications, a JFrame can be considered the outermost container for all other user interface objects.

Analyzer divides its display into two panels. One panel is of type AnalyzerTreePanel which controls the display and manipulation of groups. The other panel is of type JDesktop and is a container for multiple windows called JInternalFrames that are in turn containers for cfChart and cfReport objects. Analyzer uses the JSplitPane mechanism for dividing the display, which allows the user to move a bar along the horizontal axis to

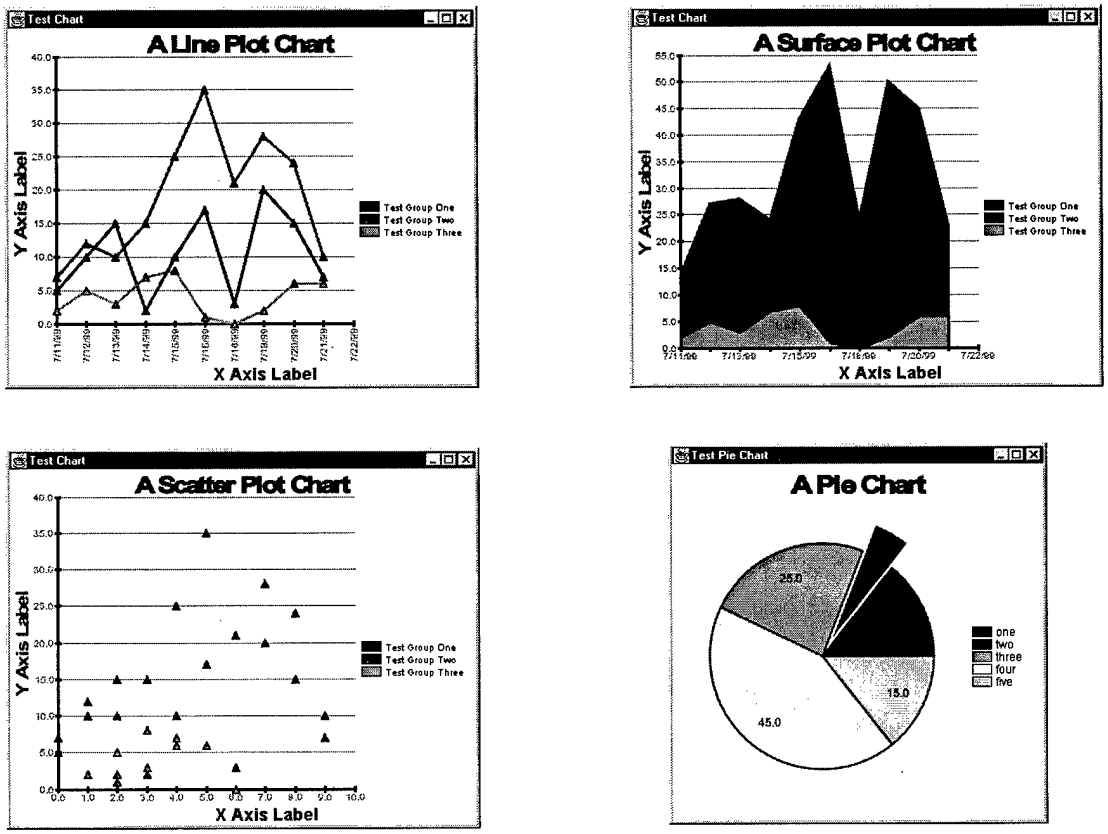


Figure 5-8. Line, Scatter, and Pie Charts

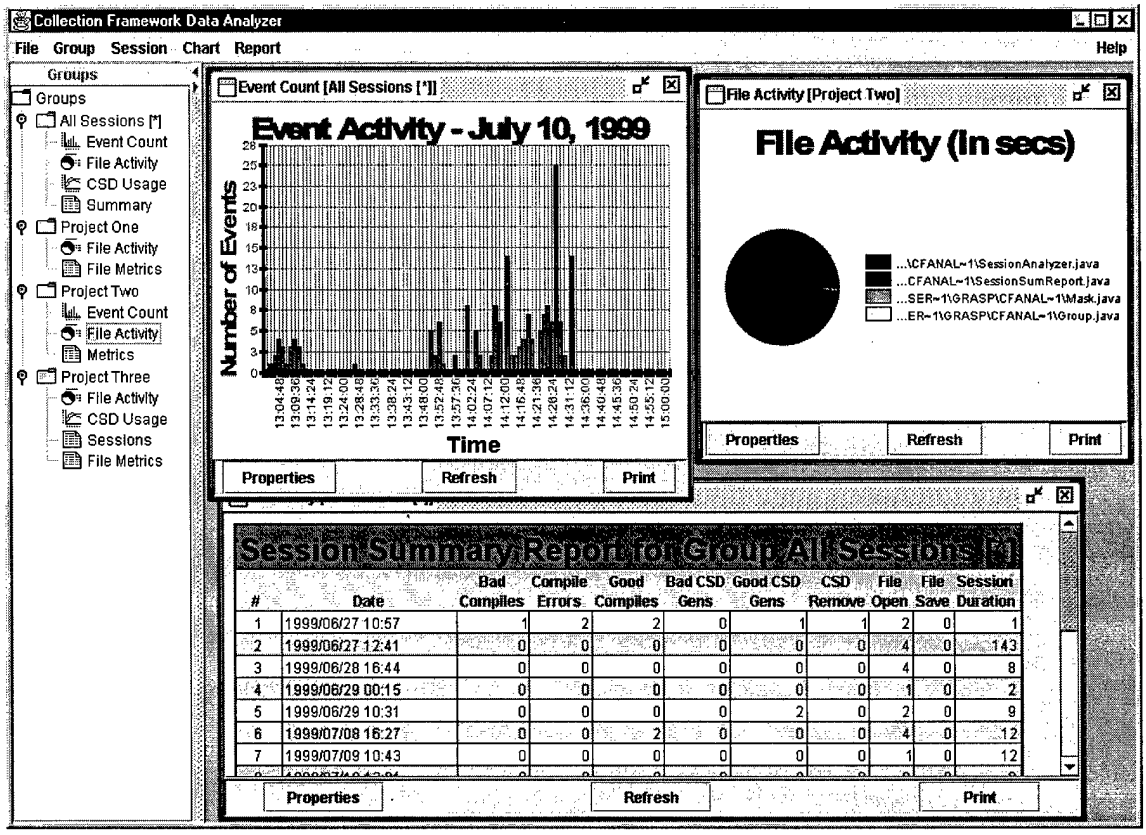


Figure 5-9. Collection Framework Analyzer

divide the displayed region. Figure 5-9 shows the main CFA display with the two panes and divider.

5.4.5.1 Group Tree Pane

As shown in Figure 5-9, the AnalyzerTreePanel shows all the groups defined. Each group shown on the tree panel is an object of type DefaultMutableTreeNode which the Java JTree structure uses to show tree diagrams. DefaultMutableTreeNode contains an object reference that CFA uses to point to a Group for group display nodes in the tree,

and to a GroupItem for leaf nodes in the tree which are the charts and reports. Depending on the type of node, a unique icon is rendered for each chart and report type.

The AnalyzerTreePanel has a mouse listener interface attached to it so that if the user right-clicks the mouse on a group or group display item, a popup menu appears which allows them to manipulate it. If the user double-clicks on a display item, it opens the corresponding chart or report in the desktop pane.

5.4.5.2 Chart/Report Desktop Pane

An important capability of CFA is to be able to compare different charts to understand the differences between groups. This is supported by the JDesktopPane portion of the main application window. This pane opens individual frames and organizes them in a manner similar to Microsoft's Multiple Document Interface (MDI). For a detailed description of MDI, see [Petzold 1996].

Each frame on the desktop pane is an instance of AnalyzerIFrame. AnalyzerIFrame subclasses JInternalFrame and adds to it two major components: the chart or report, and a set of buttons that control it. The chart or report is a panel that is returned by a call to the method getDisplayPanel() which is defined in the GroupItem interface. The control buttons allow the user to bring up a preferences dialog via the GroupItem queryProperties() method, requery the data being displayed to ensure it is current via the GroupItem refreshData() method, and print the display item. As of this writing, print functionality had not been implemented.

5.4.5.3 Groups

As noted in the design, the application loads the Groups from the database. The implementation of this is accomplished by calling the static method getDBGroupList() on the Group class. The Group class then goes to the database and instantiates instances of itself, returning the instances as a Java Vector collection.

Each Group object maintains a vector of charts and a vector of reports. It provides operations for the creation and removal of these items through the add(), removeChart(), and removeReport() methods. The methods getChartList() and getReportList() allow the object to return lists of its contained charts and reports.

Groups are responsible for removing themselves from the database. To accomplish this, the application calls a group's removeFromDB() method. This deletes any information from the database regarding this group. The group's charts and reports are also removed as part of this process.

5.4.5.4 Reports and Charts

Similar to the Group class, both Chart and Report contain static methods for loading instances from the database. The Group calls getDBChartList() and getDBReportList() in the Chart and Report classes. These classes return a Vector containing instances of themselves created from information stored in the database. Like Groups, reports and charts contain remove() methods that remove them from the database.

Reports and charts do not directly manipulate the display. Instead, when the GroupItem method getDisplayPanel() is called, they access the database to process event

information. This information is stored in an intermediate format and then loaded into an instance of cfChart or cfReport. The cfChart or cfReport instance is passed back from the getDisplayPanel() call and is eventually embedded in an AnalyzerIFrame() object.

An additional responsibility of group items is to display a preferences dialog unique to the chart or report being shown. Each chart and report has its own dialog that appears when the GroupItem method queryProperties() is called. The user changes the dialog settings, and the group item stores the settings into the database where they are remembered the next time this chart or report is opened on this particular group. An example dialog is shown in Figure 5-10.

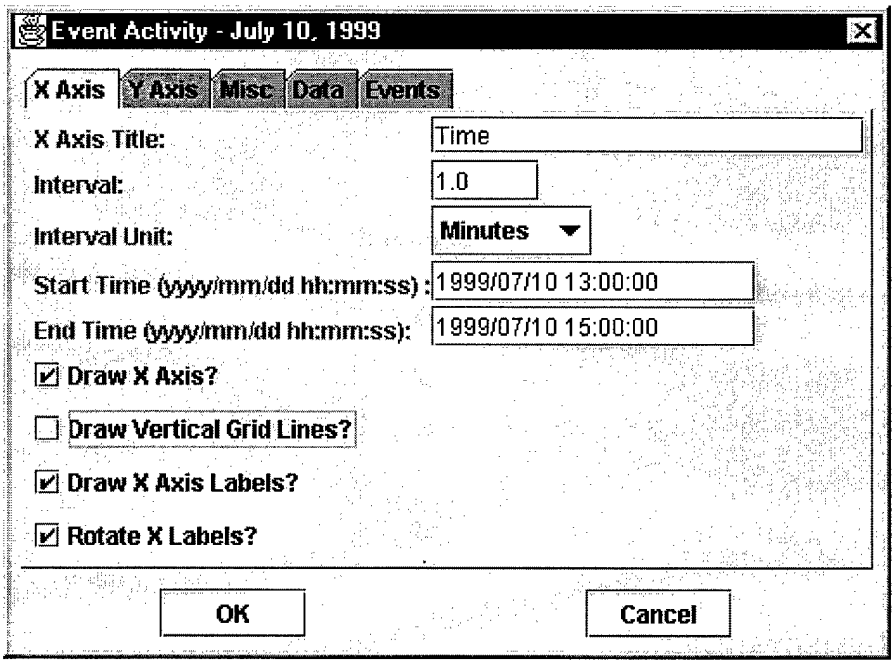


Figure 5-10. CFA Preferences Dialog

5.4.5.5 Multithreading

Several of the tasks in CFA can take a long period of time to execute. In order to give the user feedback about the progress of the task, Java's JProgressBar was used. This bar works by polling a task at specified intervals. The task being polled supplies an integer value or a string for the JProgressBar to display. The task also signals when it has finished, at which point the JProgressBar removes itself.

To make this work several key tasks in CFA had to be set up to run in their own thread. These tasks are the event file import, report generation in the Report classes, and chart generation in the Chart classes. This somewhat complicates the implementation since each object with this kind of task must implement the interface (defined by CFA, not Java) ProgressTask. Objects that implement this interface can be passed to an instance of ProgressDialog which will monitor them. ProgressDialog subclasses ProgressBar to tailor it to the CFA environment.

A minor change was made to CFDatabaseManager to support this concept. As records are read from event files and imported into the database, a counter is kept. By launching the import in a different thread, a ProgressBar is created that monitors this count of events to show progress.

5.5 TESTS CONDUCTED

To verify that the framework operated correctly under a variety of conditions, several tests were performed. These tests checked to ensure that: 1) events were recorded accurately; 2) software measures were recorded and transferred, 3) large amounts of data

would not adversely affect performance, and 4) the data was portable across platforms. Detailed results of each test are contained in Appendix B. The descriptions of each test and a summary of results are given in the following sections.

The host machine contained the following equipment:

- Processor: 400 Mhz Intel Celeron.
- Memory: 96 Megabytes of PC100 SDRAM.
- Hard Disk: 10 Gigabyte Western Digital at 5400 RPM, 11 ms average seek time.
- Motherboard: ABIT BM6 set to 66 Mhz bus speed.
- Operating System: Microsoft Windows 98 (Linux for the portability test)

5.5.1 Session Data Test

The first test was to ensure that events were being correctly recorded by the framework. A script was developed that caused all possible events to be generated. This script was executed manually in pcGRASP and the event file was then exported. The script and results are in Appendix B.

After manually checking the ASCII event file to ensure all events had been recorded in the correct format, CFA was used to import them into the database. A command-line tool supplied with InstantDB was then run and all the events were listed in the Events table to ensure they had been correctly recorded. CFA was then used to generate a session detail report and the list of events in that report was checked to ensure they were all correctly listed.

In each case the data was correctly recorded and retrieved, indicating the framework is working. An unexpected side-effect of the Windows 98 environment is that when a menu is activated, it deactivates the editing window and reactivates it when the menu is closed. This results in extra activate/deactivate actions being recorded. Since the resolution of the pcGRASP collection mechanism is in seconds, these extra events occur so quickly they do not contribute to the final totals for file activity.

5.5.2 Software Metrics Test

In this test the metrics collection routines of pcGRASP and recording/retrieval routines of the framework were tested. The goal was to ensure that logical lines of code, physical lines of code, and cyclomatic complexity were being correctly handled.

To accomplish this, three contrived Java files were created to exercise the metrics routines. The first file, "Test1.java," contained one class and all the methods were simple routines containing either no code, an if statement with one predicate, and if statement with multiple predicates, or a switch statement of vary degrees of complexity. The first file was not part of a package.

The next file, "Test2.java," was named part of a package called "metricstest" and contained two classes. The first class repeated the methods in the first file, "Test1.java." The second class contained methods with mixed if/switch constructs.

The last file, "Test3.java," was named part of the package "metricstest" and contained a single class. This class contained a method with an anonymous inner class that

contained conditions. The class itself also contains a named inner class with conditions inside its methods.

pcGRASP was used to run the metrics analyzer across the three files. CFA was then used to import the data into the database and generate a software metrics report. Both the ASCII event file generated by pcGRASP, and the report generated by CFA were then checked for consistency.

The reports produced by pcGRASP were compared against the reports extracted from the database by CFA. These reports matched, indicating the data is being stored correctly at the file, class, and method level. The reports were not checked for correctness in terms of LOC counts or cyclomatic complexity. This would require a much larger and more comprehensive suite of test programs and is beyond the scope of this research. The files used for the test are in Appendix B.

5.5.3 Large Dataset Test

This test exercised the ability of the framework to store information in the database and retrieve it when the dataset being manipulated was large. This was also a test of the capability of the underlying database, InstantDB, since the framework passes along the appropriate SQL statement depending on what operation is being performed. The author of InstantDB indicates that the database performs satisfactorily up to 1,000,000 records (the limit of his testing).

His results were verified by inserting 250,000 events into the database in 50,000 event increments and taking time measurements at each increment for time to insert, time to

# Events	DB Size	Insert Time	Rec/Sec	Full Extract	Rec/Sec	Single Extract	Rec/Sec
0	80k						
50058	13042k	1425.53	35.1	52.56	952.4	.31	522.6
100116	25599k	1780.41	28.1	100.6	995.2	.41	395.1
150174	38557k	2118.69	23.6	190.05	790.2	.55	294.5
200232	51515k	2426.34	20.6	256.47	780.8	.71	228.2
250290	64473k	2835.42	17.7	321.84	777.7	.79	205.1

Table 5-1. Database Load Test Results

generate a full session summary report, and time to generate a summary report on a set on two sessions (162 events). In addition, the size of the database tables was recorded at each increment to determine if they were growing in a linear fashion. Growth larger than linear would indicate a potential problem in how the database was storing information internally.

The 50,000 record event files were created by performing multiple concatenations of the events from the script in the session data test described previously. CFA was then instrumented to compute the time of each import operation, and the time to generate a session summary report. Note that the session summary report analyzes each event and therefore performs maximum evaluation of the data.

The results of this test are shown in Table 5-1. The time for extracting the data in summary reports is the average of five executions of each report. Full extract refers to running a report across all sessions in the database. A single extract refers to running a report on a group containing the first and last session in the database.

This test revealed several problems with InstantDB. As can be seen from the table, performance of inserts degraded as the number of records increased. In addition, both kinds of extracts suffered performance penalties as the number of records increased. These problems do not prohibit users from using InstantDB as the underlying database, but this may involve several making compromises. First, the user may need to limit the size of the database by not collecting line-level data. The activities around moving the cursor and mouse will probably account for a large majority of the events in a file. Second, the user can ensure that separate databases are maintained for different projects to prevent one database from becoming overly large. Finally, the user can substitute another database with JDBC capability and a faster database engine.

5.5.4 Platform Portability Test

The final test of the framework was to ensure that cross-platform compatibility existed. To accomplish this, the data from the first test was placed into an event database under Windows 98. This database was then copied to the same machine running Linux. CFA was then executed using the Blackdown Linux distribution of JDK 1.2. A new group with two sessions was created. A session summary report and a file activity chart was created was created.

The database was then copied back to the original Windows 98 environment. CFA was used to verify that the Linux created group and reports could be opened and examined.

The results of this test found no problems in executing CFA on all the environments. The database's integrity remained intact regardless of the environment manipulating it, as all reports showed the expected results. Based on this test, it appears that CFA and the database do have satisfactory cross-platform capabilities.

CHAPTER 6. CONCLUSIONS

6.1 INTRODUCTION

The preceding chapters discussed the problem with collecting software and human comprehension measurements. Background work in this area was presented followed by a specific set of problems that needed to be solved to meet measurement needs. A data collection framework that stored measurements as events to preserve time information was proposed as a solution. The previous chapter demonstrated how the framework could be successfully integrated into an IDE and used for capturing these measurements.

This chapter examines the conclusions that can be drawn from this work. First, the framework is evaluated to show that it did meet the goals outlined in Chapter 3. Next, the impact of the framework is evaluated in terms of what new knowledge has been gained by creating it, and what the benefits are to research and industry. Last, future work that can be conducted in this area is examined.

6.2 DOES THE FRAMEWORK MEET ITS GOALS?

Chapter 3 outlined several specific problems that needed to be solved. The following sections discuss how the data collection framework solves problems in the areas of measure correlation, data collection, data representation, storage, and analysis.

6.2.1 Correlation of Measures

The problem here was that studies of comprehension often do not take into account the nature of the code that is being used. It was demonstrated that not only can the framework track programmer actions as they work with the code, it can simulaneously record the state of the code when the actions took place. This is easily done using the events over time and software metrics over time reports.

6.2.2 Data Collection

This problem dealt with how to accurately observe and record programmer actions. Research uses this for comprehension experiments, industry for the personal software process. The framework solves this by providing a transparent mechanism for recording detailed programmer actions. This activity can then be examined in summary form via the session summary report, or in detailed form in the session detail report.

6.2.3 Data Representation

Representing the data properly so that it could be analyzed was a major issue. Not only is there difficulty in correlating measures, as mention above, but there are problems that need to be avoided in terms of using the wrong scales for storage and analysis. The framework is not able to prevent a user from analyzing the data using an incorrect scale if the analysis tool they write wants to do that. What it does do, however, is ensure that the data is not summarized on the wrong scale internally. In other words, for measures such as cyclomatic complexity which Fenton argues is nominal, the framework does not store

it internally as averaged values over time. Instead, explicit values for each point in time are stored, avoiding incorrect nominal scale use.

6.2.4 Storage and Analysis

The last major problem involved creating a historical database of file activity and metrics. The framework keeps all of its data in a relational database. Through reports such as file activity industry users can examine this data to predict how long future projects will take. Additionally, since the framework tracks cyclomatic complexity of code, managers can also predict the testing burden for new projects based on previous data.

6.3 IMPACT OF THE FRAMEWORK

A key criteria of any research project such as this is to examine whether it expands the body of knowledge in its field, and whether that knowledge produces tangible benefits. In this work, new information has been contributed to the field in terms of a methodology for: 1) how to correlate software and human experiments, and 2) how to correlate those experiments with time. The framework is the tool that implements that correlation methodology.

The following sections discuss the major benefits to this approach.

6.3.1 Benefits to Research

Researchers looking into how programmers comprehend software can now follow their actions long term. Most experiments cited have been of short duration, usually just a few hours, where the subjects were observed and questioned during the experiment. With the

framework, they can now be monitored for long periods to determine what files they used and when they used them to gauge whether they were adopting a bottom-up approach or a top-down approach. This is not to imply the framework is a necessarily a substitute for observation, there will be a need to occasionally ask the programmer why they took certain actions. The framework could improve accuracy, however, if combined with direct observation.

Projects that are developing new techniques for visualizing or manipulating software will gain significant benefits. The framework can answer questions such as: How often are the features used, and for how long? Is the feature used throughout development, or only at certain critical points? What was the nature of the software? By understanding the context of when a feature was activated, researchers can look at ways to improve its usefulness.

Those attempting to validate software metrics in terms of real-world use will find the framework very beneficial. The framework keeps a historical database of a system's metrics, and this can be used to correlate with reported errors. Also, the framework can answer questions like: Does more complex software require more maintenance time? Does more complex software result in more errors or changes over the life of the software?

6.3.2 Benefits to Industry

Industry gains from the framework's ability to track historical information regarding what set of files is being used the most often. If a manager notices that a file has activity

associated with it that is far out of proportion to other files, it may mean the file should be examined to determine if it needs to be split into smaller segments. It may mean the file is a candidate for being rewritten if the activity can be correlated to testing faults associated with the file.

Additionally, the framework can point out which files have not just had the most activity in terms of time spent on them, but it can also indicate files that have had significant structural changes over time. If a file changes significantly in terms of its lines of code, complexity, or other software measure, this may indicate a need to examine it in a code walkthrough.

Finally, the framework's ability to track time spent in different phases of development will help those individual programmers who are logging their time as part of PSP.

6.4 FUTURE WORK

The fields of software comprehension and validation of software metrics are constantly looking at difficult problems and probing into new avenues of research. This leaves a lot of room for expanding the framework and using it to further other research goals.

6.4.1 Framework Expansion

The framework as written is a core set of components that can be easily expanded. Enhancements that would provide significant benefits include:

- Integration of the framework into other environments such as CASE tools.
- Integration of the framework with the Java version of GRASP.

- Adding enhanced reports to assist users with the computations involved in PSP.
- Inclusion of comprehension test scores as stored data to show how comprehension changed over time in long-term experiments.
- Implement the ability to print all charts and reports.
- Implement the ability to export charts and reports to HTML format.
- Create a network server that feeds events to an instance of CFFileManager for real-time data collection. This would also require changes to CFInterface to allow it to send events to the network server process instead of directly to a file manager.

6.4.2 Research

The capabilities of the framework allow researchers to look in greater depth at some key areas:

- *CSD Studies*. The framework can be used to record long-term studies of CSD usage to determine how and when programmers use it.
- *Architectural Diagrams*. GRASP will soon have the ability to show architectural diagrams. Studies using the framework can determine what diagrams are used most often, and at what points in development they used.
- *Structure Folding*. Another feature being added to GRASP, structure folding allows users to "fold" code so that it is hidden from view. The framework can be used in studies to determine how users take advantage of this feature.

- *Correlation Studies.* An interesting research project would be to recreate the studies of von Mayrhauser, Soloway, Pennington using a variety of code structures to determine what effect different software attributes have on comprehension strategies. The framework would be essential to knowing if a programmer switches strategies as the code complexity changes.

The framework provides a valuable tool in the researcher and industry user's toolkit of data collection methodologies. Its ability to transparently collect data for long-term studies means that it is now possible to create experiments to accurately gauge the effects of different visualizations. In the next step of research, beyond this dissertation, the framework and CFA will find direct uses as they are put to work in gauging the effectiveness of the CSD on program development.

REFERENCES

ALBRECHT, A.J. 1979. Measuring Application Development Productivity. *Proceedings of the Joint SHARE, GUIDE, and IBM Application Development Symposium*, (October 1979), 83-92.

ALBRECHT, A.J. 1984. *AD/M Productivity Measurement and Estimate Validation*, IBM Corporate Information Systems, IBM Corp., Purchase, N.Y.

ANTIS, J.M., EICK, S.G., AND PYRCE, J.D. 1996. Visualizing the Structure of Large Relational Databases. *IEEE Software*, (January 1996), 72-79.

ARUNACHALAM, V., AND SASSO, W. 1996. Cognitive Processes in Program Comprehension: An Empirical Analysis in the Context of Software Engineering. *Journal of Systems Software*, 34, 177-189.

BALL T., AND EICK, S.G. 1996. Software Visualization in the Large. *Computer*, 18, 4 (April 1996), 33-43

BASILI, V.R., AND HUTCHENS, D.H. 1983. An Empirical Study of a Syntactic Complexity Family. *IEEE Transactions on Software Engineering*, 9(6), 664-672.

BASILI, V.R., AND WEISS, D.M. 1984. A Methodology for Collecting Valid Software Engineering Data. *IEEE Transactions on Software Engineering*, SE-10, 6 (November 1984), 728-738.

BASILI, V.R., BRIAND, L.C., AND MELO, W.L. 1996. A Validation of Object-Oriented Design Metrics as Quality Indicators. *IEEE Transactions on Software Engineering*, 22, 10 (October 1996), 751-761.

BATES, P.C. 1995. Debugging Heterogeneous Distributed Systems Using Event-Based Models of Behavior. *ACM Transactions on Computer Systems*, 13, 1 (February 1995), 1-31.

BIEMAN, J.M. 1996. Fundamental Issues in Software Measurement. *Software Measurement*, International Thomson Computer Press, London, U.K.

BOEHM, B.W., BROWN, J.R., AND KASPAR, J.R. 1978. *Characteristics of Software Quality*, TRW Series of Software Technology, Amsterdam, North Holland.

BOEHM-DAVIS, D.A., FOX, J.E., AND PHILIPS, B.H. 1996. Techniques for Exploring Program Comprehension. *Proceedings of the 1996 Workshop on Empirical Studies of Programmers*, 3-37.

BOOCH, G. 1994. *Object-Oriented Analysis and Design with Applications*, Benjamin/Cummings, Redwood City, CA.

BRIAND, L.C., MORASCA, S., AND BASILI, V.R. 1996. Property-Based Software Engineering Measurement. *IEEE Transactions on Software Engineering*, 22, 1 (January 1996), 68-85.

CAMPIONE, M., AND WALRATH, K. 1998. *The Java Tutorial Second Edition: Object-Oriented Programming for the Internet*, Addison-Wesley, Reading, MA.

CANFORA, G.L., MANCINI, L., AND TORTORELLA, M. 1996. A Workbench for Program Comprehension During Software Maintenance. *Proceedings of the 4th Workshop on Program Comprehension*, 30-39.

CARPENDALE, M.S.T., COWPERTHWAITE, D.J., AND FRACCHIA, F.D. 1997. Extending Distortion Viewing from 2D to 3D. *IEEE Computer Graphics and Applications*, (July/August 1997), 42-51.

CHAN, P., MUNRO, M. 1997. PUI: A Tool to Support Program Understanding. *Proceedings of the 5th International Workshop on Program Comprehension*, 192-198.

CHIDAMBER, S.R., AND KEMERER, C.F. 1994. A Metrics Suite For Object Oriented Design. *IEEE Transactions on Software Engineering*, 20, 6 (June 1994), 476-493.

CITRIN, W.C., SANTIAGO, C., AND ZORN, B. 1996. Scalable Interfaces to Support Program Comprehension. *Proceedings of the 4th Workshop on Program Comprehension*, 123-132.

COAD, P., YOURDON, E. 1991a. *Object-Oriented Analysis*, Yourdon Press, Englewood Cliffs, NJ.

COAD, P., YOURDON, E. 1991b. *Object-Oriented Design*, Yourdon Press, Englewood Cliffs, NJ.

CORNELL, G., AND HORSTMANN, C. S. 1997, *Core Java, Second Edition*, SunSoft Press/Prentice Hall, Upper Saddle River, NJ.

CROSS, II J.H., BAROWSKI, L.A., HENDRIX, T.D., AND TEATE, J.C. 1996. Control Structure Diagrams for Ada95, In *TRI-Ada '96, Conference Proceedings on Disciplined Software Development in Ada*, 143-147.

CROSS II, J.H., CHANG, K.H., HENDRIX, T.D., CHAPMAN, R.O., AND MCQUAID, P.A. 1997. Visualization and Measurement of Source Code. *Crosstalk,* STSC, Hill Air Force Base, Utah, 10, 12 (December 1997), http://www.stsc.hill.af.mil/CrossTalk/1997/dec/visualization.html

CURTIS, B., SOLOWAY, E.M., BROOKS, R.E., BLACK, J.B., EHRLICH, K., AND RAMSAY, H.R. 1996. Software Psychology: The Need for an Interdisiplinary Program, *Proceedings of the IEEE*, 74, 8 (August 1986).

CURTIS, B., SHEPPARD, S.B., KRUESI-BAILEY, E., BAILEY, J., BOEHM-DAVIS, D.A. 1989. Experimental Evaluation of Software Documentation Formats. *The Journal of Systems and Software*, 9, 167-207.

DEMARCO, T. 1979. *Structured Analysis and System Specification*, Prentice-Hall, Englewood Cliffs, NJ.

EICK, S.G., STEFFEN, J.L., AND SUMMER, E.E. 1992. Seesoft – A Tool For Visualizing Line Oriented Software Statistics. *IEEE Transactions on Software Engineering*, 18, 11 (November 1992), 957-968.

FAVRE, J. 1997. Understanding-In-The-Large. In *Proceedings of the 5th International Workshop on Program Comprehension*, 29-38.

FEDCHAK, E., AND VIENNEAU, R. 1996. A History of Software Measurement at Rome Laboratory. November 1996, Technical Report, Contract F30602-89-C-0082, Rome Laboratory, RL/C3C, Rome, New York.

FENTON, N. 1994. Software Measurement: A Necessary Scientific Basis. *IEEE Transactions on Software Engineering*, 20, 3 (March 1994), 199-206.

FENTON, N., PFLEEGER, S.L., AND GLASS, R.L. 1994. Science and Substance: A Challenge to Software Engineers. *IEEE Software*, (July 1994), 86-95.

FENTON, N., AND PFLEEGER, S.L. 1997. *Software Metrics: A Rigorous & Practical Approach*, International Thomson Computer Press, London, UK.

FIUTEM, R., MERLO, E., ANTONIOL, G., AND TONELLA, P. 1996. Understanding the Architecture of Software Systems. In *Proceedings of the 4th Workshop on Program Comprehension*, 187-196.

FIX, V., WIEDENBECK, S., AND SCHOLTZ, J. 1993. Mental Representations of Programs by Novices and Experts. In *Proceedings of INTERCHI '93*, (April 1993), 74-79.

FURNAS, G.W. 1986. Generalized Fisheye Views. *Proceedings of CHI '86*, ACM, New York, 16-23.

GERSHON, N., AND EICK, S.G. 1995. Visualization's New Tack: Making Sense of Information. *IEEE Spectrum*, (November 1995), 38-56.

GRADY, R.B, AND CASWELL, D.L. 1987. *Software Metrics: Establishing a Company-Wide Program,* Prentice-Hall, Englewood Cliffs, NJ.

GRADY, R.B. 1992. *Practical Software Metrics for Project Management and Process Improvement*, Prentice-Hall, Englewood Cliffs, NJ.

HALL, T., AND FENTON, N. 1997. Implementing Effective Software Metrics Programs. *IEEE Software*, (March/April 1997), 55-64.

HALSTEAD, M. 1997. *Elements of Software Science*, Elsevier Science Press.

HAROLD, E.R. 1997. *Java Network Programming*. O'Reilly & Associates, Inc, Sebastopol, CA.

HARTSON, H.R., CASTILLO, J.C., KELSO, J., KAMLER, J., AND NEALE, W.C. 1996. Remote Evaluation: The Network as an Extension of the Usability Laboratory. In *Proceedings, Conference on Human Factors in Computing Systems (CHI '96)*, 228-235.

HENDERSEN-SELLERS, B. 1996. *Object-Oriented Metrics: Measures of Complexity*, Prentice-Hall, Englewood Cliffs, NJ.

HENDRIX, T.D., CROSS II, J.H., BAROWSKI, L., AND MATHIAS, K.S. 1998. Providing Enhanced Visual Support for Software Development and Maintenance. *Proceedings of the 36th Annual ACM Southeast Conference*, Marietta, GA, (April 1998), 23-28.

HERITAGE 1989. *The American Heritage Dictionary*, Dell Publishing.

HILBERT, D., AND REDMILES, D. 1998. Agents for Collecting Application Usage Data Over The Internet. In *Proceedings, Second International Conference on Autonomous Agents (Agents '98)*.

HORSTMANN, C.S., AND CORNELL, G. 1997. *Core Java 1.1 , Volume 1 – Fundamentals*, Prentice Hall, Englewood Cliffs, NJ.

HUDEPOHL, J.P., AUD, S.J., KHOSHGOFTAAR, T.M., ALLEN, E.B., AND MAYRAND, J. 1996. Emerald: Software Metrics and Models on the Desktop. *IEEE Software*, (September 1996), 56-60.

HUMPHREY, W.S. 1997. *Introduction to the Personal Software Process*. Addison-Wesley.

JONES, C. 1991. *Applied Software Measurement*, McGraw Hill.

JONES, C. 1996a. Programming Languages Table. Software Productivity Research, Inc., Online Article, March 1996, <http://www.spr.com/library/0langtbl.htm>

JONES, C. 1996b. Our Worst Current Development Practices. *IEEE Software*, (March 1996), 102-104.

KAN, S.H. 1995. *Metrics and Models in Software Quality Engineering*, Addison-Wesley.

KAY, J., AND THOMAS, R.C. 1995. Studying Long-Term System Use, *Communications of the ACM*, 38, 7 (July 1995), 61-69.

KITCHENHAM, B., PFLEEGER, S.L., AND FENTON, N. 1995. Towards a Framework for Software Measurement Validation. *IEEE Transactions on Software Engineering*, 21, 12 (December 1995), 929-943.

KORTH, H.F., AND SILBERSCHATZ, A. 1991, *Database System Concepts*, McGraw-Hill Inc., New York, NY.

KRUGLINSKI, D.J. 1996, *Inside Visual C++*, Microsoft Press, Redmond, WA.

LEDGARD, H.F. 1975. *Programming Proverbs*, Hayden Book Company, NJ.

LEUNG, Y.K., AND APPERLY, M.D. 1994. A Review and Taxonomy of Distortion-Oriented Presentation Techniques. *ACM Transactions on Computer-Human Interaction*, 1, 2 (June 1994), 126-160.

LINOS, P., AUBET, P., DUMAS, L., HELLEBOID, Y., LEJEUNE, P., AND TULULA, P. 1993. Facilitating the Comprehension of C Programs: An Experimental Study. *Proceedings of the 2nd Workshop on Program Comprehension*, Capri, Italy, (July 1993), 78-86.

LORENZ, M., AND KIDD, J. 1994. *Object-Oriented Software Metrics*, Prentice-Hall, Englewood Cliffs, NJ.

MACKINLAY, J.D., ROBERTSON, G.G., AND CARD, S.K. 1991. The Perspective Wall: Detail and Content Smoothly Integrated, In *Proceedings CHI '96*. 173-179.

MCCABE, T.J. 1976. A Complexity Measure. *IEEE Transactions on Software Engineering*, SE-2, 4 (December 1976), 308-320.

MCCABE, T.J., AND BUTLER, C.W. 1989. Design Complexity Measurement and Testing. *Communications of the ACM*, 32, 12 (December 1989), 1415-1425.

MCCALL, J., RICHARDS, P., AND WALTERS, G. 1977. Factors in Software Quality. RADC-TR-77-369, Vol I – III, November 1977. NTIS AD-A049-014, 015, 055.

MCQUAID, P. 1996. *Profiling Software Complexity*. Ph.D Dissertation, Auburn University, Auburn, AL, August 1996.

MIARA, R.J., MUSSELMAN, J.A., NAVARRO, J.A., AND SHNEIDERMAN, B. 1983. Program Indentation and Comprehensibility. *Communications of the ACM*, 26, 11 (November 1983), 861-867.

NASSI, I., AND SHNEIDERMAN, B. 1973. Flowchart Techniques for Structured Programming. *ACM SIGPLAN Notices*, 8, 8, 12-26.

NEAL, A. S., AND SIMONS, R. M. 1983. Playback: A Method for Evaluating the Usability of Software and its Documentation. In *Proceedings, Conference on Human Factors in Computing Systems (CHI '83)*, 78-82.

OFFEN, R.J., AND JEFFREY, R. 1997. Establishing Software Measurement Programs. *IEEE Software*, (March/April 1997), 45-53.

PENNINGTON, N. 1987. Comprehension Strategies in Programming. *Empirical Studies of Programmers: Second Workshop*, Ablex Publishing Corporation, Norwood, NJ, 100-113.

PERRY, D.E., STAUDENMAYER, N.A., AND VOTTA, L.G. 1994. People, Organizations, and Process Improvement. *IEEE Software*, (July 1994), 35-45.

PETZOLD, C. 1996, *Programming Windows 95*, Microsoft Press, Redmond, WA.

PFLEEGER, S.L. 1997. Status Report on Software Measurement. *IEEE Software*, (March/April 1997), 33-43.

PRESSMAN, R. 1997. *Software Engineering: A Practitioners Approach*, McGraw-Hill.

RATIONAL CORPORATION 1997. UML Specification Version 1.1. Online document available at www.rational.com/uml.

ROSS, N. 1987. The Collection and Use of Data for Monitoring Software Projects. In *Measurement for Software Control and Assurance*, B.A. Kitchenham and B. Littlewood, editors, Elsevier Science Publishers, Essex, U.K.

RUMBAUGH, J., BLAHA, M., PREMERLANI, W., EDDY, F., AND LORENSEN, W. 1991. *Object-Oriented Modeling and Design*, Prentice-Hall, Englewood Cliffs, NJ.

SARKAR, M., AND BROWN, M.H. 1994. Graphical Fisheye Views. *Communications of the ACM*, 37, 12 (December 1994), 73-84.

SHEPPERD, M., AND INCE, D. 1993. *Derivation and Validation of Software Metrics*, Oxford Science Publications.

SIDARKEVICUTE, D., TYUGU, E., AND KUUSIK, A. 1996. A Knowledge-Based Toolkit for Software Visualization. *Proceedings of Knowledge Based Software Engineering 1996.*

SIOCHI, A.C., AND EHRICH, R.W. 1991. Computer Analysis of User Interfaces Based on Repetition in Transcripts of User Sessions, *ACM Transactions on Information Systems*, 9, 4 (October 1991), 309-335.

SOLOWAY, E., BONAR, J., AND EHRLICH, K. 1983. Cognitive Strategies and Looping Constructs: An Empirical Study. *Communications of the ACM*, 26, 11 (November 1983), 853-860.

SOLOWAY, E. 1986. Learning to Program = Learning to Construct Mechanisms and Explanations. *Communications of the ACM*, 29, 9 (September 1986), 850-858.

SPC 1995. (The Software Productivity Consortium), *The Software Measurement Guidebook*, International Thomson Computer Press, London, UK.

STOREY, M.-A.D., FRACCHIA, F.D., AND MÜLLER, H.A. 1997. Cognitive Design Elements to Support the Construction of a Mental Model during Software Visualization. *Proceedings of the 5th International Workshop on Program Comprehension*, 17-28.

TAPP, R., AND KAZMAN, R. 1994. Determining the Usefulness of Colour and Fonts in a Programming Task. *Proceedings of the 3rd Workshop on Program Comprehension*, Washington, D.C., 154-161.

TEATE, J. 1998. *An Analysis of the Use of the GRASP Software Engineering Environment and the Control Structure Diagram*, Masters Degree Thesis, Auburn University, Alabama, April 1998.

WEYUKER, E.J. 1988. Evaluating Software Complexity Measures. *IEEE Transactions on Software Engineering*, 14, 9 (September 1988), 1357-1365.

VON MAYRHAUSER, A., AND VANS, A.M. 1993. From Program Comprehension to Tool Requirements for an Industrial Environment. *Proceedings of the 2nd Workshop on Program Comprehension*, Capri, Italy, (July 1993), 78-86.

VON MAYRHAUSER, A., AND VANS, A.M. 1994. Dynamic Code Cognition Behaviors for Large Scale Code. *Proceedings of the 3rd Workshop on Program Comprehension*, Washington, D.C., (November 1994), 74-81.

VON MAYRHAUSER, A., AND VANS, A.M. 1996a. Identification of Dynamic Comprehension Processes During Large Scale Maintenance. *IEEE Transactions on Software Engineering*, 22, 6, 424-437.

VON MAYRHAUSER, A., AND VANS, A.M. 1996b. On the Role of Hypotheses during Opportunistic Understanding While Porting Large Scale Code. *Proceedings of the 4th Workshop on Program Comprehension*, 68-77.

WILDE, N., AND CASEY, C. 1996. Early Field Experience with the Software Reconnaissance Technique for Program Comprehension. *Proceedings of the 3rd Working Conference on Reverse Engineering*, 312-318.

YOURDON, E.N., AND CONSTANTINE, L.L. 1978. *Structured Design*, Yourdon Press.

ZUSE, H. 1997. *A Framework of Software Measurement*, Walter de Gruyter & Co., Berlin, Germany.

APPENDIX A. ASCII EVENT FILE FORMAT

All events in an ASCII file consist of a set of fields starting with a timestamp, followed by the event type. Additional fields define key characteristics of each event. While any white space characters may be inserted between fields, the tab character is the most efficient since it only requires one character and leaves the resulting file human-readable.

Legend:

<ts> = Timestamp. Seconds elapsed since Jan 1st, 1970.

<wh> = Window Handle. Window event occurred in.

<logLOC> = Logical Lines of Code.

<phyLOC> = Physical Lines of Code.

CFEvtAlterLine:	<ts>	1	<wh>	<line #>	
CFEvtBadCompile:	<ts>	2	<wh>	<# of errors>	
CFEvtCSDBadGen:	<ts>	3	<wh>	<line #>	
	<text of error message>				
CFEvtCSDGoodGen:	<ts>	4	<wh>		
CFEvtCSDOff:	<ts>	5	<wh>		
CFEvtDeleteLine:	<ts>	6	<wh>	<start #>	<# lines>
CFEvtEnterLine:	<ts>	7	<wh>	<line #>	

CFEvtExitLine:	<ts>	8	<wh>	<line #>	
CFEvtFileActive:	<ts>	9	<wh>		
CFEvtFileInactive:	<ts>	10	<wh>		
CFEvtFileClose:	<ts>	11	<wh>		
CFEvtFileOpen:	<ts>	12	<wh>	<filename>	
CFEvtFold:	<ts>	13	<wh>	<start #>	<# lines>
CFEvtGoodCompile:	<ts>	14	<wh>		
CFEvtInsertLine:	<ts>	15	<wh>	<start #>	<# lines>
CFEvtInterruptOff:	<ts>	16			
CFEvtInterruptOn:	<ts>	17			
CFEvtPhase:	<ts>	18	<phase>		
CFEvtSessionEnd:	<ts>	19			
CFEvtSessionStart:	<ts>	20			
CFEvtUnfold:	<ts>	21	<wh>	<start #>	<# lines>
CFEvtFileSave:	<ts>	26	<wh>	<filename>	
CFEvtFileNew:	<ts>	27	<wh>		
CFEvtCSDOn:	<ts>	28	<wh>		
CFEvtMetricsReport:	<ts>	29	<language>	<wh>	

The metrics report initial line is followed by subsequent lines in one of the following formats:

File Metrics:	1	<logLOC>	<phyLOC>

Class Metrics:	2	<logLOC>	<phyLOC>	
Method Metrics:	3	<logLOC>	<phyLOC>	<complexity>
End of Report:	0			
CFEvtExecuteFile:	<ts>	30	<wh>	

APPENDIX B. TEST SCRIPTS, FILES, AND RESULTS

B.1 DATA TEST.

The data test checked to ensure that pcGRASP was properly recording events to its event file and that the framework was properly storing them in the database. The following test script was executed to generate the event file:

Step	Action	Expected Result
1	Start pcGRASP.	SessionStart event written
2	Create new Java file (designate this Window 1).	FileNew event written FileActive event written InsertLine at line 1 event written EnterLine at line 1 event written
3	Open previously created Java file "TestPlan1.java". (designate Window 2). Record time.	FileOpen event written FileInactive for Window 1 written FileActive event for Window 2 written EnterLine at line 1 event written
4	Click on Window 1, line 1.	FileInactive for Window 2 written FileActive for Window 1 written
5	Enter the following source code: **public class TestPlan** { **public static void main(String [] args)** { **System.out.println("TestPlan")** **// Delete** } }	a) AlterLine for line 1 b) InsertLine for line 2 c) EnterLine for line 2 d) AlterLine for line 2 (repeat b,c,d for lines 3,4,5,6,7,8)
6	Save file as "TestPlan.java"	FileSave event with name "TestPlan.java" written
7	Click "Compile"	BadCompile event written with 1 error
8	Click "Generate CSD"	BadCSDGen event written
9	Click on line numbers button, add	EnterLine event for line 5

Step	Action	Expected Result
	semicolon to end of line5.	AlterLine event for line 5
10	Click "Save"	FileSave event with name "TestPlan.java" written
11	Click "Compile"	GoodCompile event written
12	Click "Generate CSD"	GoodCSDGen event written
13	Click "Run"	FileExecution event written
14	Delete line 6 by highlighting it and then pressing the DEL key followed by BACKSPACE	DeleteLine event for line 6
15	Click "Save As" from menu. Save as "TestPlan2.java"	FileSave event with name "TestPlan1.java" written.
16	Wait 60 seconds	(Leaves time gap to show TestPlan1 was active during this time)
17	Close "TestPlan2.java" in Window 1	FileClose event on Window 1 FileActive event on Window 2
18	Wait 120 seconds	(Leave time gap to show some activity)
19	Exit pcGRASP via Exit on the File menu.	FileClose event on Window 1 SessionEnd event
20	Start pcGRASP	SessionStart event written FileOpen event for "TestPlan1.java" CSDOn event for "TestPlan1.java" FileActive event for "TestPlan1.java" EnterLine event for "TestPlan1.java"
21	Wait 60 seconds.	(Leave time gap to show some activity)
22	Exit pcGRASP via Exit on the File menu.	FileClose event SessionEnd event

The resulting event file follows:

```
936283441   20
936283459   27      0
936283459   9       0
936283474   9       0
936283474   10      0
936283474   9       0
936283474   10      0
936283475   12      1       F:\mathias\dissertation\diss\Test1.java
936283475   9       1
936283475   28      1
```

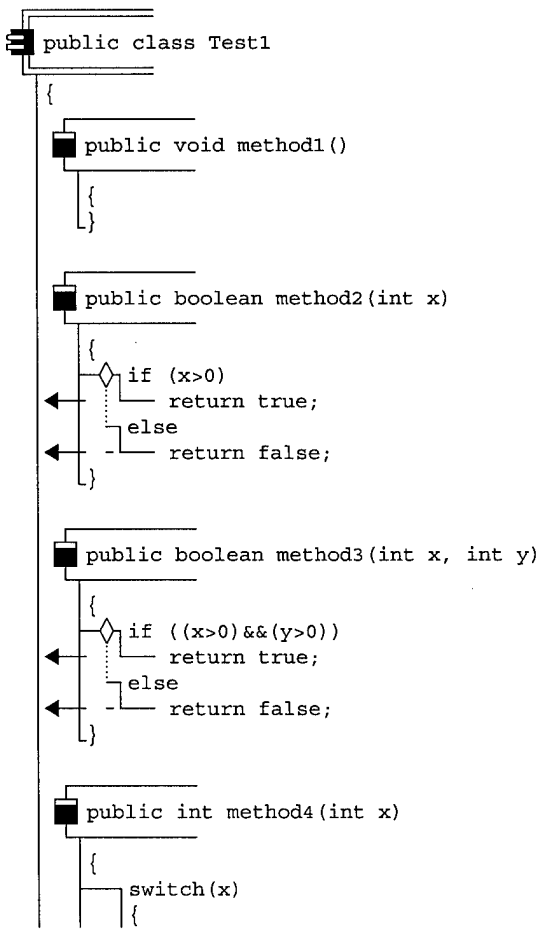
```
936283479   10      1
936283479   9       0
936283483   1       0       1
936283488   15      0       2       1
936283488   7       0       2
936283489   1       0       2
936283491   15      0       3       1
936283491   7       0       3
936283491   1       0       3
936283501   15      0       4       1
936283501   7       0       4
936283502   1       0       4
936283502   15      0       5       1
936283502   7       0       5
936283503   1       0       5
936283513   15      0       6       1
936283513   7       0       6
936283513   1       0       6
936283517   15      0       7       1
936283517   7       0       7
936283518   1       0       7
936283519   15      0       8       1
936283519   7       0       8
936283520   1       0       8
936283536   9       0
936283536   10      0
936283536   9       0
936283536   26      0       F:\mathias\dissertation\diss\TestPlan.java
936283540   26      0       F:\mathias\dissertation\diss\TestPlan.java
936283540   26      1       F:\mathias\dissertation\diss\Test1.java
936283544   2       0       1
936283547   9       0
936283547   3       0
F:\mathias\dissertation\diss\TestPlan.java:7  Grasp: Syntax (parse) error at or before
"}".
936283557   7       0       5
936283559   1       0       5
936283562   26      0       F:\mathias\dissertation\diss\TestPlan.java
936283565   26      0       F:\mathias\dissertation\diss\TestPlan.java
936283565   26      1       F:\mathias\dissertation\diss\Test1.java
936283569   14      0
936283572   4       0
936283576   30      0
936283576   10      0
936283576   10      0
936283576   9       0
936283586   7       0       6
936283589   1       0       6
936283591   1       0       5
936283592   6       0       6       1
936283592   7       0       5
936283603   9       0
936283603   10      0
936283603   9       0
936283603   26      0       F:\mathias\dissertation\diss\TestPlan2.java
936283666   11      0
936283666   10      0
936283666   9       1
936283666   10      0
936283787   10      1
936283787   11      1
936283787   10      1
936283787   10      1
936283787   19
936283793   20
```

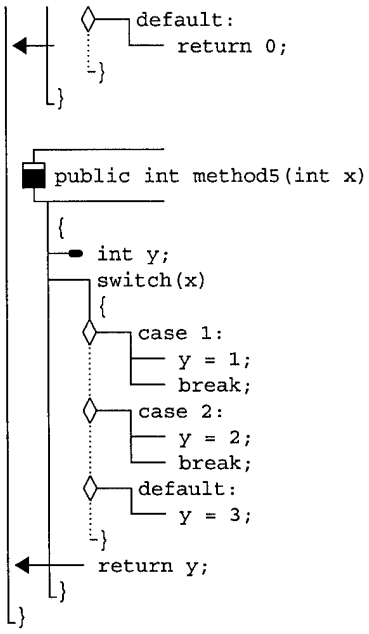
```
936283793   12      0       F:\mathias\dissertation\diss\Test1.java
936283793   9       0
936283793   28      0
936283857   10      0
936283857   11      0
936283857   10      0
936283857   10      0
936283857   19
```

B.2 METRICS COLLECTION TEST

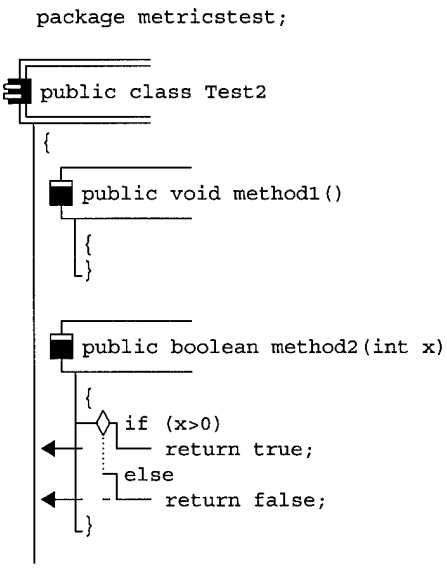
The metrics test was designed to ensure that the metrics data collected and written to an event file, was properly stored in the framework database. To exercise this, the following files were run through the metric generation routines:

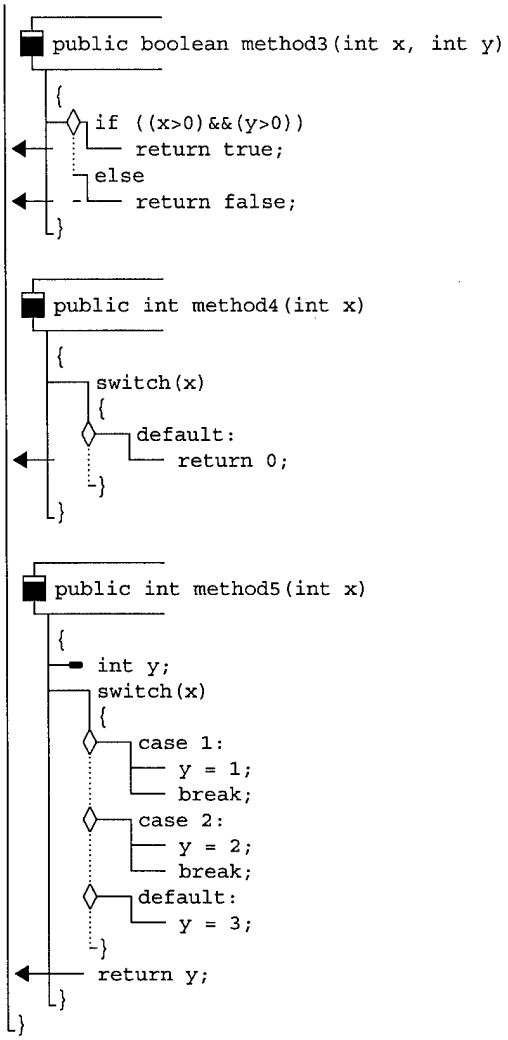
File One:

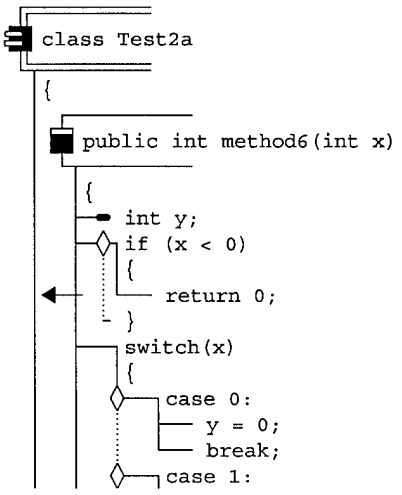


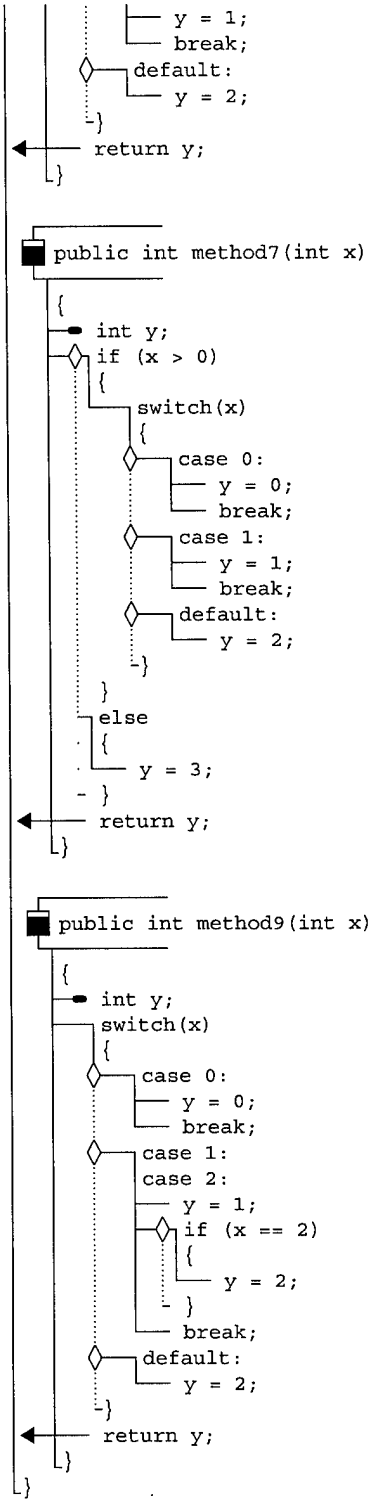

default:
return 0;
}
}
public int method5(int x)
{
int y;
switch(x)
{
case 1:
y = 1;
break;
case 2:
y = 2;
break;
default:
y = 3;
}
return y;
}
}

File Two:

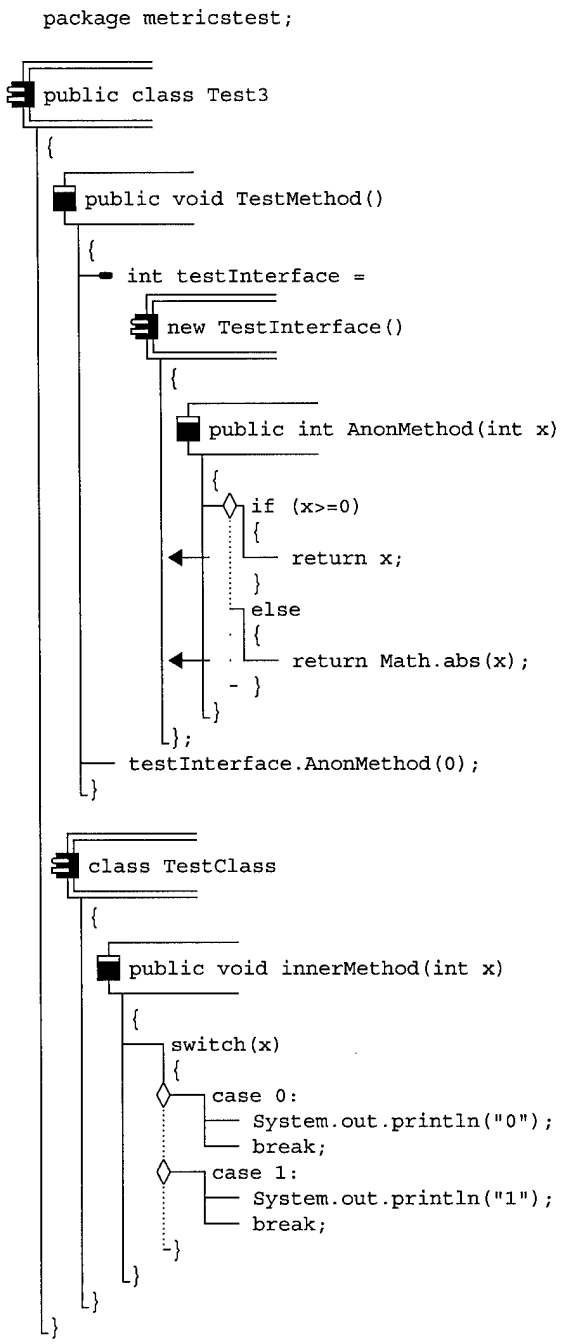

package metricstest;
public class Test2
{
public void method1()
{
}
public boolean method2(int x)
{
if (x>0)
return true;
else
return false;
}

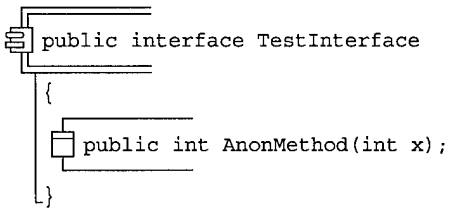

public boolean method3(int x, int y)
{
if ((x>0)&&(y>0))
return true;
else
return false;
}
public int method4(int x)
{
switch(x)
{
default:
return 0;
}
}
public int method5(int x)
{
int y;
switch(x)
{
case 1:
y = 1;
break;
case 2:
y = 2;
break;
default:
y = 3;
}
return y;
}
}


class Test2a
{
public int method6(int x)
{
int y;
if (x < 0)
{
return 0;
}
switch(x)
{
case 0:
y = 0;
break;
case 1:


y = 1;
break;
default:
y = 2;
}
return y;
}
public int method7(int x)
{
int y;
if (x > 0)
{
switch(x)
{
case 0:
y = 0;
break;
case 1:
y = 1;
break;
default:
y = 2;
}
}
else
{
y = 3;
}
return y;
}
public int method9(int x)
{
int y;
switch(x)
{
case 0:
y = 0;
break;
case 1:
case 2:
y = 1;
if (x == 2)
{
y = 2;
}
break;
default:
y = 2;
}
return y;
}
}

File Three:


package metricstest;
public class Test3
{
public void TestMethod()
{
int testInterface =
new TestInterface()
{
public int AnonMethod(int x)
{
if (x>=0)
{
return x;
}
else
{
return Math.abs(x);
}
}
};
testInterface.AnonMethod(0);
}
class TestClass
{
public void innerMethod(int x)
{
switch(x)
{
case 0:
System.out.println("0");
break;
case 1:
System.out.println("1");
break;
}
}
}
}



B.3 DATA LOAD TEST

The data load test exercised the ability of the framework and database to process large numbers of events. The following information is the raw data recorded to compute the results shown in Table 5-1.

# Events	DB Size	Insert Time	Full Extract	Single Extract
50058	13042k	1425.53	66.19	.49
			48.44	.27
			53.82	.28
			45.04	.22
			49.31	.31
100116	25599k	1780.41	114.03	.38
			96.34	.44
			97.44	.44
			96.56	.39
			98.62	.41
150174	38557k	2118.69	174.39	.55
			167.25	.54
			160.82	.55
			270.18	.55
			177.59	.54
200232	51515k	2426.34	272.76	.72
			307.03	.66
			224.76	.66
			229.53	.72
			248.27	.71
250290	64473k	2835.42	301.43	.83
			418.86	.82
			315.00	.77
			289.30	.77
			284.63	.77